UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Dejan Golja

ORKESTRACIJA IN RAZPOREJANJE VSEBNIKOV V VISOKO RAZPOLOŽLJIVIH SISTEMIH

MAGISTRSKO DELO

Mentorica: doc. dr. Mojca Ciglarič

Ljubljana, 2016

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Večna pot 113
1000 Ljubljana, Slovenija
telefon: 01 47 98 100
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Številka: 160-MAG-ISO/2016
Datum: 06. 04. 2016

Dejan GOLJA, univ. dipl. inž. rač. in inf.

L j u b l j a n a

Fakulteta za računalništvo in informatiko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **Orkestracija in razporejanje vsebnikov v visoko razpoložljivih sistemih**

Orchestration and scheduling of containers in highly available environments

Tematika naloge:

Vsebniki (ang. container) so po funkcionalnosti podobni virtualnim strojem, saj omogočajo, da na fizičnem strežniku znotraj navideznih operacijskih sistemov hkrati poganjamo več aplikacij. Prednosti prinašajo predvsem v sistemih, kjer so zahtevani visoka razpoložljivost in izkoriščenost sistemskih virov, standarizacija okolja, hitrejši vzpostavitveni čas in lažje razporejanje aplikacij med različnimi vozlišči. Pri tem pa omogočajo tudi enostavno postavitev razvojnega okolja.
Raziščite, kako to tehnologijo najbolje uporabiti v produkcijskih okoljih, kjer morajo biti poleg lažje uporabe zadovoljene tudi ostale zahteve, kot so varnost, izolacija virov, visoka razpoložljivost in sledljivost spremembam. Osredotočite se na večstrežniška okolja, v katerih poganjamo vsebnike z enakimi, dolgoročno tekočimi storitvami, ki morajo biti za končne uporabnike transparentno in neprekinjeno dostopne prek enotnih vstopnih točk. Preučite, kako vzpostaviti orkestracijski sistem, ki samodejno usklajuje, upravlja in skrbi za ponovno vzpostavitev vsebnikov (in s tem storitve) v primeru izpada.,.
Primerjajte orodja in tehnologije, ki omogočajo napredno orkestracijo in razporejanje vsebnikov. Sistem, ki temelji na vsebnikih, primerjajte s konkretnim obstoječim sistemom, ki temelji na virtualnih strojih, in utemeljite smiselnost odločitve za prehod na vsebnike.

M e n t o r i c a :

M. Ciglarič

doc. dr. Mojca Ciglarič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

D e k a n :

prof. dr. Nikolaj Zimic

Zahvala

Zahvaljujem se mentorici, doc. dr. Mojci Ciglarič, za vse nasvete in pomoč, ne le ob izdelavi magistrskega dela, temveč tudi tekom celotnega študija.

Hvala Markotu za potrpljenje in pomoč pri urejanju mojih misli skozi ves moj študij.

Zahvala gre tudi sodelavcem podjetja Fairfax Media iz Avstralije in Nove Zelandije za vso pomoč in potrpežljivost pri razumevanju določenih vidikov naše infrastrukture.

Posebna zahvala pa gre mojim staršem in moji partnerici, ki so mi vedno stali ob strani.

Kazalo

1. Uvod ... **3**

1.1 Opis problema ... **4**

1.2 Nameni in cilji dela ... **6**

1.3 Pregled področja ... **8**

1.4 Metodologija ... **10**

2. Tehnologija vsebnikov in orodja za orkestracijo ... **12**

2.1 Tehnologija vsebnikov ... **13**

2.1.1 Docker ... 15

2.1.1.1 Omrežna povezanost ... 17

2.2 Kubernetes ... **19**

2.2.1 Komponente ... 20

2.2.2 Arhitektura gruče ... 21

2.3 *Apache Mesos* ... **23**

2.3.1 Komponente ... 23

2.3.2 Arhitektura ... 24

2.4 *Docker Swarm* ... **26**

2.4.1 Arhitektura ... 27

2.5 Primerjava orodij po funkcionalnosti ... **29**

3. Meritve ... **35**

3.1 Opis pilotnega okolja ... **35**

3.2 Rezultati testov ... **38**

3.2.1 Izkoriščenost virov ... 38

3.2.2 Prepustnost sistema ... 40

3.2.3 Čas, potreben za samodejno odkrivanje storitev ... 42

3.2.4 Čas, potreben za nadgradnjo vsebnikov ali aplikacije ... 43

3.2.5 Izolacija virov ... 45

3.2.6 Dodajanje novih vozlišč v gručo ... 46

3.2.7 Čas obnovitve storitve po izpadu delovanja ... 48

3.3 Rezultati ... **49**

4. Predlog rešitve za medijsko hišo ... **50**

4.1 12-faktorska aplikacijska zasnova ... **55**

4.1.1 Koda ... 56

4.1.2 Odvisnosti ... 56

4.1.3 Konfiguracijske datoteke ... 56

4.1.4 Podporne storitve ... 57

4.1.5 Zgradi, objavi, poženi ... 57

4.1.6 Procesi ... 59

4.1.7 Povezava vrat ... 59

4.1.8 Sočasnost ... 59

4.1.9 Enkratna uporaba procesa 60
4.1.10 Enakost predprodukcijska/produkcijska okolja 61
4.1.11 Dnevniški zapisi 61
4.1.12 Proces upravljanja 61
5. Sklepne ugotovitve 62
6. Viri 65

Kazalo slik

Slika 1-1: Arhitektura vsebnikov v primerjavi s klasično namestitvijo aplikacije na gostitelju 3
Slika 1-2: Diagram konkurentov našim storitvam 4
Slika 2-1: Poenostavljena arhitektura Docker na enem vozlišču 15
Slika 2-2: Omrežna povezanost *Docker* na način mosta 18
Slika 2-3: Gostiteljev način omrežja *Docker* 19
Slika 2-4: Kubernetes arhitektura 21
Slika 2-5: Osnovna arhitektura *Mesos* 24
Slika 2-6: Poenostavljen diagram *DC/OS Apache Mesos* 25
Slika 3-1: Diagram tipične aplikacije *Fairfax* v oblaku *AWS* 36
Slika 4-1: Diagram elementov *Kubernetes* 51
Slika 4-2: Arhitektura namestitve *Fairfax Media Kubernetes* v oblaku *AWS* 53
Slika 4-3: Natanča namestitev okolja *Kubernetes* v *AWS* 54
Slika 4-4: Elementi različnih faz življenjskega cikla aplikacije 58
Slika 4-5: Procesni model povečanja prepustnosti 60

Kazalo tabel

Tabela 2-1: Primerjava Kubernetes, Mesos in Swarm po lastnostih 30
Tabela 3-1: Uporaba sistema v primeru mirujočega sistema 39
Tabela 3-2: Čas za samodejno odkrivanje nove storitve 43
Tabela 3-3: Čas za nadgradnjo aplikacije za primer dveh kopij storitve 44
Tabela 3-4: Vzpostavitveni časi za novo vozlišče 47
Tabela 3-4: Čas obnovitve storitve po izpadu 48

Seznam kratic in pojmov

AWS	*Amazon Web Services* (Amazonove spletne storitve)
AWS ASG	*AWS Autoscaling Group* (Samodejna skupina)
AWS ELB	*AWS Elastic Load Balancer* (Elastičen razporejevalec prometa)
AWS EBS	*AWS Elastic Block Storage* (Elastično blokovsko polje)
AWS AZ	*AWS Availability Zone* (Dostopno območje)
Docker Hub	*Docker Public Image Repository* (Repozitorij za nalaganje in snemanje javno dostopnih slik)
SLA	Software Service Agreement (pogodba o delovanju storitve)
VCS	*Version Control System* (Sistem za upravljanje s kodo)
K8s	*Kubernetes* (Urejevalnik gruč vsebnikov *Kubernetes*)

Povzetek

Trenutno smo v obdobju pospešenega uvajanja vsebnikov – iz meseca v mesec vse več podjetij uvaja ali raziskuje možnosti uporabe te tehnologije v svojih okoljih. Kljub temu, da tehnologija in orodja, povezana z uporabo le-te, še niso standardizirana, prinašajo veliko prednosti. Omogočajo poenostavljen razvoj, poenostavi uporabo mikrostoritev in hitro prilagajanje zmogljivosti glede na promet. Kljub temu sami vsebniki niso dovolj za produkcijska okolja, kjer imamo veliko strožje zahteve, kot so visoka razpoložljivost, tolerantnost do napak, samodejno odpravljanje napak itd. Zato smo se odločili, da bomo preučili še spremljevalna orkestracijska orodja, ki uporabljajo vsebnike kot centralni del tehnologije. Glede na potrebe našega podjetja smo predstavili trenutno najbolj popularna orkestracijska orodja. Primerjali smo jih s konkretnim obstoječim sistemom, ki temelji na virtualnih strojih v oblaku, in utemeljili smiselnost odločitve za prehod na vsebnike.

V magistrskem delu smo najprej pregledali tehnologije vsebnikov in predstavili *Docker* kot najbolj razširjeno implementacijo vsebnikov. Nato smo predstavili trenutno najbolj zanimiva orkestracijska orodja, in sicer *Mesos*, *Swarm* in *Kubernetes*. Za tem smo izvedli teste delovanja in primerjali rezultate z obstoječo rešitvijo v oblaku. Orkestracijska orodja samo tudi primerjali po funkcionalnosti. Na koncu smo predstavili našo izbrano arhitekturno rešitev za medijsko hišo. Na koncu pokaženo, da je naša rešitev po izvedbi in funkcionalnosti primerna.

Ključne besede

vsebniki, *Docker*, *Kubernetes*, *Mesos*, *Swarm*, visoko razpoložljivi sistemi, orkestracija

Abstract

We are in a period of accelerated adoption of containers - every month more and more companies are introducing and exploring the possibility of using this technology in their environment. Despite the fact that the technology and tools associated with its use are not yet fully standardized, they still bring a lot of benefits. They enable simplified development, empower the usage of microservices and provide a better platform to adjust the capacity of the resources, based on traffic. Nevertheless, containers themselves are not enough for production environments where there are strict requirements such as high availability, fault tolerance, self-healing, etc. That is why we also examined the current orchestration tools, using containers as a central part of the technology. Based on the needs of our company, we picked the most suitable orchestration tools. We compared them with our existing system hosted in the cloud, and justified the decision to switch to containers.

In this thesis, we first reviewed the containers technology, mostly focusing on Docker as the most popular implementation at the moment. Then we presented and analyzed the current orchestration tools - Mesos, Swarm and Kubernetes. After that we run performance tests and compared the results with performance of our existing system. We also compared the orchestration tools based on their functionality. Finally, we suggested the architectural solution suitable for the company and presented a pilot implementation. We show that the pilot meets the functional and performance requirements.

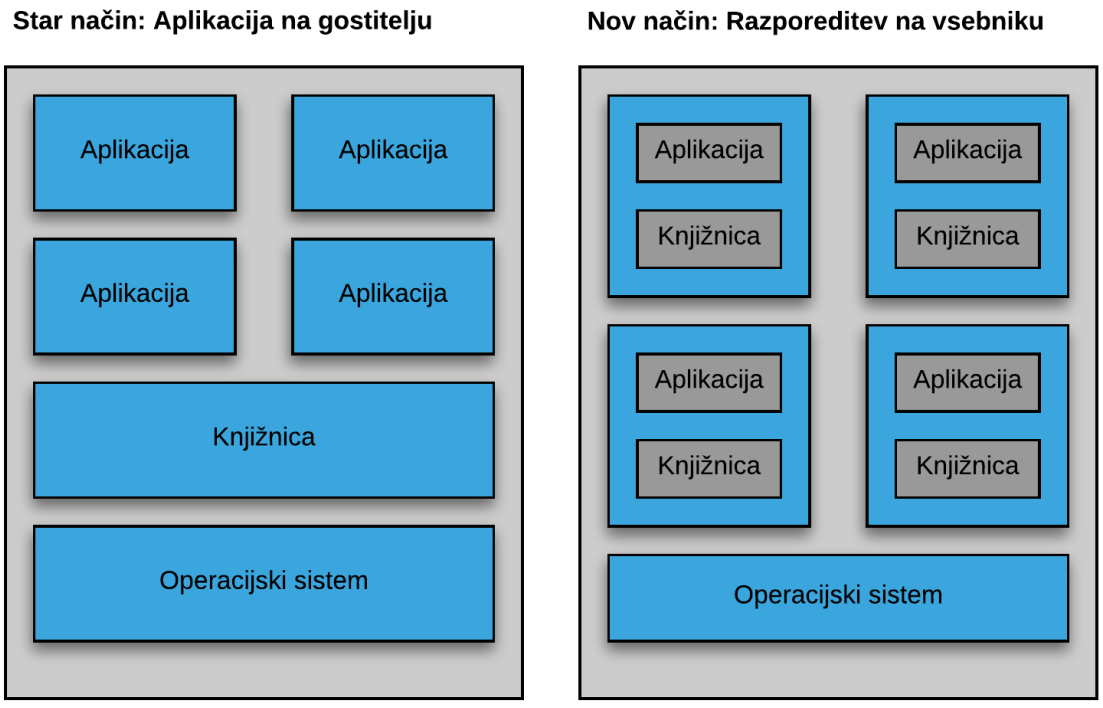
Keywords

containers, Docker, Kubernetes, Mesos, Swarm, orchestration, high availability

1. Uvod

V času pisanja magistrskega dela smo v obdobju pospešenega uvajanja vsebnikov – iz meseca v mesec vse več podjetij uvaja ali raziskuje možnosti uporabe te tehnologije v svojih okoljih [1, 2, 3]. Kljub temu, da je koncept vsebnikov že dalj časa prisoten v UNIX-ovih okoljih [4], je bila dosedanja uporaba omejena le na večja podjetja, predvsem zaradi kompleksnosti rešitve in posledično ekonomske neupravičenosti implementacije. Za povečanje popularnosti vsebnikov se imamo zahvaliti predvsem podjetju Docker Inc., ki je leta 2013 izdelalo orodje, ki omogoča enostavno uporabo te tehnologije, in poslovnemu modelu ponudnikov v oblaku, ki zaračunavajo ceno na osnovi zakupljenih virov.

Iz sistemskega vidika vsebniki omogočajo uporabo enake aplikacijske slike v pred-produkcijskih in produkcijskih okoljih, kar je zelo pomembno za stalno integracijo (ang. *continuous integration*), zmanjševanje razlik med okolji in pogoj za 12-faktorsko metodologijo razvoja aplikacij (ang. *twelve-factor app methodology*) [5, 6].



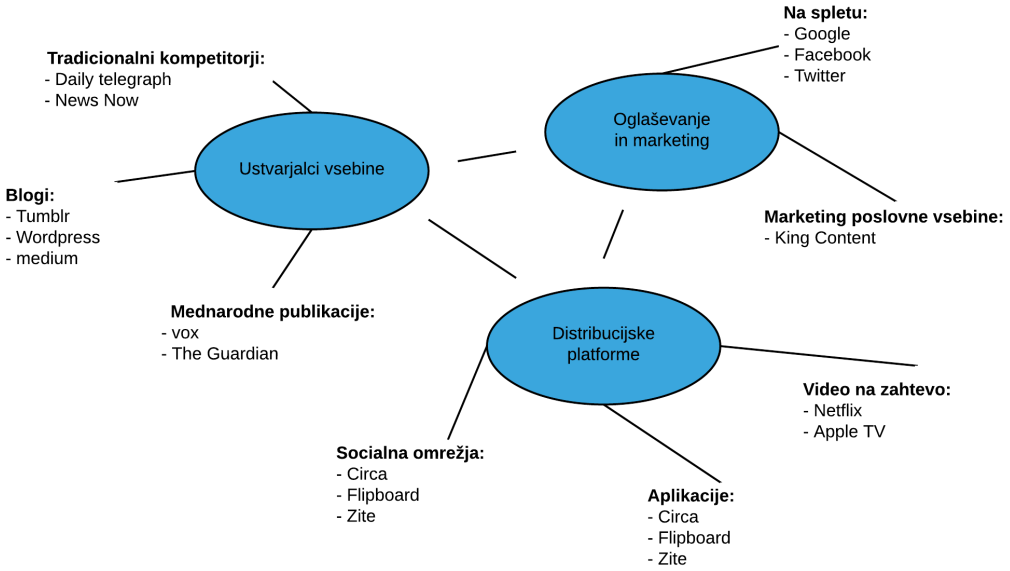
Slika 1-1: Arhitektura vsebnikov v primerjavi s klasično namestitvijo aplikacije na gostitelju

Kot je razvidno iz slike 1-1, je mogoče s pomočjo vsebnikov na enostavnejši način izolirati sistemske zahteve in hkrati poganjati različne aplikacije, ki imajo morda tudi konfliktne zahteve na skupnem operacijskem sistemu.

Glavni cilji uvedbe vsebnikov v primerjavi s samo virtualizacijskimi tehnologijami so povečanje izkoriščenosti računalniških virov, poenostavitev namestitve aplikacij in zagotavljanje visoke razpoložljivosti storitev s poenostavljeno stalno integracijo [7].

1.1 Opis problema

V podjetju, kjer sem zaposlen, vedno skušamo optimizirati naše procese s ciljem, izboljšati storitve in racionalizirati stroške. Kot največja medijska hiša v Avstraliji in Novi Zelandiji, ki ponuja nekaj najbolj obiskanih spletnih strani z novicami v državi, imamo veliko različnih nehomogenih sistemov in storitev. Tradicionalno je bilo zelo težko vstopiti v posel tiskanega medija, saj je bila začetna investicija v opremo zelo visoka in tvegana, kar pa danes ne velja več. Danes tiskane časopise nadomeščajo digitalne različice, kar pomeni vedno več konkurence s strani lokalnih publikacij in prav tako s strani globalnih IKT-podjetij, ki tradicionalno niso bili naši konkurenti, ampak so zaradi želje po povečanju obiskanosti in razumevanja navad njihovih uporabnikov začeli posegati po novičarskem trgu. Primeri takih produktov so *Google Reader, Facebook Instant Articles* in družabna omrežja za novinarje, kot je, denimo, *Medium* (prikazano na sliki 1-2).



Slika 1-2: Diagram konkurentov našim storitvam

Zgodaj smo ugotovili, da lastništvo lastnega podatkovnega centra ekonomsko ni več smotrno, saj so se naše potrebe po računalniških virih nenehno spreminjale, kar ni bilo mogoče zadovoljiti s tradicionalno infrastrukturo in najemnimi pogodbami dobaviteljem strojne opreme. Imeli smo težave s pomanjkanjem fizičnega prostora, premalo ali preveč strojne opreme itd. Večino najemnih in podpornih pogodb je bilo mogoče skleniti le za večletno obdobje, kar je zelo otežilo načrtovanje in razvoj naših prihodnjih storitev. Naše podjetje je zato eno izmed prvih medijskih hiš v Avstraliji, ki je vse svoje storitve preselilo v oblak in izrabilo prednosti računalniških virov na zahtevo in podatkovnih baz kot storitev. Prednost so vsekakor fleksibilna plačljiva shema in standarizirani programski vmesniki, ki omogočajo definiranje ponovljive infrastrukture s pomočjo kode (ang. *infrastructure as a code*).

Kljub vsem prednostim smo še vedno imeli težave z neoptimalno uporabo virov pri uporabi klasičnih virtualnih strojev. Zapletene storitve onemogočajo hitro samodejno prilagajanje povečanemu prometu ob izrednih dogodkih in izjemno povečanem prometu. Čas, ki je potreben za zagon dodatnega virtualnega stroja in namestitev aktualne programske opreme, je prevelik, nehomogeni sistemi in storitve pa zaradi nezdružljivih programskih knjižic onemogočajo souporabo skupnih virtualnih strojev. Vse to povzroča več zakupljenih računalniških virov, kot bi bilo treba (in s tem povezane nepotrebne stroške), nekateri strežniki so neaktivni in dinamično razporejanje med zakupljenimi vozlišči glede na povpraševanje ni mogoče.

Poleg same izboljšave infrastrukture in ekonomičnih prednosti, povezanih s tem, smo tudi želeli izboljšati proces upravljanja z našimi informacijskimi sistemi, saj smo ugotovili, da je proces izdaje nove različice v produkcijo predolg in predrag. Težava niso bili manjši informacijski sistemi in storitve, saj smo glede na količino prometa in število uporabnikov lahko tolerirali manjše izpade delovanja in funckionalne napake. Težave so nam prinašali predvsem bolj kompleksni sistemi, pri čemer je bilo delovanje le-teh neposredno povezano z dobičkonosnostjo podjetja. V povprečju smo za bolj kompleksne sisteme potrebovali najmanj mesečni cikel razvoja, s tem, da je bila tretjina časa uporabljenega za regresijske teste, sam postopek nagradnje sistema na novo različico pa je lahko trajal tudi nekaj dni. Tako dolgi postopki so bili potrebni predvsem zaradi zagotavljanja kvalitete sistema.

Treba se je zavedati, da imamo poleg zunanjih uporabnikov naših informacijskih sistemov tudi nekaj sto novinarjev, razpršenih po regiji APAC, ki nenehoma posodabljajo vsebino prek enotnega sistema za upravljanje vsebin, kar služi tudi kot podporni sistem za tiskani časopis. To pomeni, da lahko ob daljšem izpadu sistema pride tudi do zamude roka oddaje v tisk, iz česar sledi, da tiskane inačice časopisa ne

moremo izdati naslednji dan. V takem primeru lahko govorimo o milijonski dnevni škodi.

Posledica tega je velik poslovni pritisk na nemoteno delovanje sistema, kar se odraža tudi na tehničnem nivoju, saj postopki načrtovanja, razvoja in izdaje novih funkcionalnosti potekajo predolgo in preveč konzervativno. Posredno to vpliva tudi na uspešnost podjetja, saj se včasih prepočasi odzovemo na zahteve trga.

Dodaten problem, ki bi ga želeli v podjetju rešiti s pomočjo vsebnikov, je zagotavljanje zadovoljivega in ekonomičnega razvojnega okolja za skupino več kot dvesto razvijalcev. Simuliranje produkcijskega okolja na toliko različnih delovnih postajah predstavlja zelo velik izziv. Vsebnike želimo uporabiti za poenotenje razvojnega in produkcijskega okolja, povečanje izkoriščenosti danih virov in olajšanje procesa razvoja, testiranja in objave novih različic storitev.

Zaradi zgoraj omejenih problemov smo določili delovno skupino s ciljem, najti rešitve oz. izboljšave:

- Povečati izkoriščenost virov s pomočjo konsolidacije strežnikov in posledično zmanjšati stroške.
- Omogočiti hitrejšo vzpostavitev novih okolij.
- Omogočiti razvijalcem samodejno postavitev standardnih razvojnih okolij brez pomoči sistemskih inženirjev in hkrati omogočiti neodvisno izdajanje nove kode brez intervencije operaterjev.
- Izboljšati postopek stalne integracije in izdaje novih inačic, kar bi omogočilo hitrejšo izvedbo novih sprememb na informacijskih sistemih.
- Izvedba in kvaliteta storitev morata biti enaki oz. primerljivi s trenutnim okoljem in mehanizmi za vzpostavitev paralelnega okolja ob izpadu.

1.2 Nameni in cilji dela

Namen magistrskega dela je pokazati, kako lahko (če sploh lahko) z vsebniki rešujemo zgoraj opisane težave, obenem pa v podjetju zagotovimo enako ali boljšo kvaliteto storitev, kot to omogoča trenutna arhitektura. Upravičiti moramo tudi čas, porabljen za implementacijo in migracijo na novo rešitev.

V magistrskem delu bomo obravnavali trenutno najbolj obetavne tehnologije, ki rešujejo problem orkestracije in razporejanja vsebnikov za namen dolgoročnih storitev in preostale spremljevalne storive, ki omogočajo neprekinjeno delovanje sistema. Analizirali bomo primere uporabe, kjer vsebniki vsebujejo po en proces na sliko (ang. *RPPC - Related Process Per Container [8]*), saj so taki gradniki najbolj primerni za mikrostoritve. Osredotočili se bomo na večvozliščna okolja, na proces samodejnega odkrivanja in visoke razpoložljivosti storitev, na izmenjavo tajnih podatkov (ključi, gesla, idr.) in nadzorovanje izkoriščenosti virov.

Cilji magistrskega dela so izbrati primerna orodja za orkestracijo, vzpostaviti pilotna okolja, izbrati metrike za vrednotenje in primerjavo različic ter na osnovi rezultatov priti do priporočila in rešitve za podjetje, v katerem delam.

Želimo, da tehnologije v novem okolju omogočajo razvoj aplikacij v skladu z 12-faktorsko metodologijo razvoja aplikacij [5], ki jo je novembra 2011 sestavil soustanovitelj Herokuja Adam Wiggins.

Manifest določa:

- Koda (ang. *Codebase*) – Koda mora biti hranjena v enem sistemu za upravljanje s kodo (ang. *VCS*).

- Odvisnosti (ang. *Dependencies*) – Eksplicitno deklariraj in izoliraj odvisnosti sistema.

- Konfiguracijske datoteke – Hrani vse namestitvene vrednosti kot sistemske spremenljivke in nikoli v sistemu za upravljanje s kodo.

- Podporne storitve (ang. *Backing Services*) – Obravnavaj podporne sisteme kot dodatne vire.

- Zgradi, objavi, poženi (ang. *build, release, run*) – Strogo loči proces prevajanja, pakiranja in poganjanja aplikacije.

- Procesi (ang. *Processes*) – Izvajaj sistem kot eden ali več neodvisnih procesov.

- Povezava vrat (ang. *Port Binding*) – Izvozi storitve neposredno prek vrat brez dodatnih odvisnosti, kot so sistem *SYS V* itd.

- Sočasnost (ang. *Concurrency*) – Povečaj prepustnost s pomočjo procesnega modela.

- Enkratno uporabni (ang. *Disposability*) – Povečaj robustnost sistema s pomočjo hitrega zagona in kontrolirane zaustavitve sistema.

- Predprodukcijska/produkcijska pariteta (ang. *Development/Production Parity*) – Skušaj imeti razvojno, predprodukcijsko in produkcijsko okolje čim bolj podobno.

- Dnevniški zapisi (ang. *Logs*) – Obravnavaj dnevniške zapise kot vir dogodkov.

- Proces upravljanja (ang. *Admin Process*) – Poganjaj naloge, povezane z administracijo sistema kot del storitve.

Naš cilj je tudi izbrati najbolj primerno skupino orodij, s katero bomo lahko v največji meri izpolnili zgoraj naštete zahteve in hkrati kjer bo potrebno najmanj razvijali interna orodja, da dosežemo naše cilje. Kljub temu, da imamo v podjetju na razpolago veliko razvijalcev, smo v praksi ugotovili, da je razvoj internih orodij zelo drag, saj je težko zagotoviti dolgoročna sredstva za vzdrževanje. Zaradi omenjenega smo se odločili uporabiti obstoječo odprtokodno rešitev in po potrebi dopolniti le-to, kjer je le mogoče.

Na koncu bi radi tudi praktično prikazali, da je z uporabo vsebnikov in z njimi povezanih tehnologij in orodij možno doseči večjo izkoriščenost računalniških virov, obenem pa je to v teh danih okvirih najbolj ekonomična rešitev za storitveno medijsko podjetje.

1.3 Pregled področja

V nadaljevanju bomo strnjeno predstavili definicije najpogosteje uporabljenih pojmov v mojem delu in članke, ki preučujejo podobno problematiko.

Študije [9, 10, 11,12, 13] kažejo, da je uporaba tehnologije vsebnikov iz vidika porabe sistemskih virov vsaj enako ali bolj učinkovita kot uporaba klasičnih virtualnih strojev. W. Felter et al., ki v članku [14] predstavijo rezultate obremenitvenega testiranja virtualnih strojev *KVM* [15] in vsebnikov Docker [16], so prišli do zaključka, da "rezultati nakazujejo, da prinaša uporaba vsebnikov v primerjavi z virtualnimi stroji primerljive ali boljše zmogljivosti v skoraj vseh preizkušenih primerih". Te rezultate pripisujejo predvsem dejstvu, da v primeru vsebnikov virtualizacija poteka na nivoju Linuxovega jedra [17] [18].

Prav tako avtorji članka [19, 4] ugotovljajo, da vsebniki pomagajo zmanjševati kompleksnost okolij, poenostavijo samodejno prilagajanje povečanemu prometu, zboljšajo prenosljivost aplikacij in povečajo stabilnost sistemov. V članku Evalution of Docker Containers Based on Hardware Utilisation [20] tudi ugotovijo, da je sistemska poraba vsebnikov primerljiva z klasičnimi sistemi.

V literaturi obstaja veliko različnih definicij pojma orkestracija, saj je definicija odvisna od konteksta uporabe. Za naš primer je najustreznejša definicija avtorjev [21], in sicer "*orkestracija je programska komponenta, ki upravlja celoten življenjski cikel oblačnih aplikacij*". Isti avtorji [21] definirajo tudi oblačne aplikacije kot distribuirane aplikacije, ki jih sestavljajo večnivojske, med sabo povezane komponente. Primer takih aplikacij je npr. aplikacija blog B, ki je sestavljena iz treh komponent, in sicer usmerjevalca prometa, aplikacijskega strežnika, ki vključuje tudi poslovno logiko, in podatkovne baze.

V članku Large-Scale cluster management at Google with Borg [7] avtorji opišejo težave, s katerimi so se soočali pri orkestraciji, in odločitve pri načrtovanju orodja, ki je omogočalo upravljanje z veliko količino vsebnikov, ter način, kako so s tem izboljšali izkoriščenost virov in posredno povečali upravičenost naložbe.

Za določitev kriterijev primernosti orkestracijskih orodij, smo se zgledovali po klasifikaciji, opisani v članku *Cloud Orchestration Features: Are Tools Fit for Purpose ?* [21], v katerem avtorji določijo primernost orkestracijskega orodja na podlagi dveh večjih kriterijev, in sicer glede na nabor funkcionalnosti, povezanih s podporo namestitve orodja med različnimi oblačnimi ponudniki, in glede na lastnosti samega orodja. V našem primeru smo se bolj zgledovali po kriterijih lastnosti orodij, saj je zaradi geografske lokacije podjetja trenutno za nas primeren le en ponudnik storitve v oblaku.

1.4 Metodologija

Za identifikacijo najprimernejše rešitve smo določili nabor pogojev, ki smo jih razdelili v dve skupini, in sicer so to merljivi kriteriji in nabor funcionalnosti orodij, ki nam omogoča dosego ciljev. Za namen magistrskega dela bomo postavili tudi pilotna okolja za testiranje in primerjavo vseh izbranih orodij.

Za merljive kriterije smo izbrali:

- Izkoriščenost virov v odstotkih – Za meritev uporabljenosti bomo uporabili metodologijo, opisano v [20] in Amazonovo storitev *CloudWatch*.
- Prepustnost sistema – Merili bomo največje število transakcij na minuto, ki jih sistem zmore obdelati.
- Samodejno odkrivanje storitev na večvozliščnih namestitvah – Primerjali bomo čas, potreben za zaznavanje nove vstopne točke in za posodobitev le-te.
- Časovna zahtevnost procesa nadgradnje vsebnikov – Merili bomo čas, potreben za namestitev nove inačice programske opreme; čas, ko je bila storitev nedosegljiva; in povprečni odzivni čas sistema med nadgradnjo.
- Obnovitev storitve po izpadu delovanja – Simulirali bomo različne nepričakovane dogodke, kot so na primer prekinjena povezava med vozlišči, strojna okvara vozlišča in interno nedelovanje aplikacije. Merili bomo čas, ki je potreben, da orkestracijski sistem prepozna napako in vzpostavi normalno delovanje storitve. Prav tako bomo merili odzivni čas storitve med izpadi delov sistema.
- Izolacija virov – Simulirali bomo porazdeljeno ohromitev storitve (ang. *DDoS*) A in merili povprečen odzivni čas storitve B.
- Dodajanje novih vozlišč v gručo – Merili bomo potrebni čas za dodajanje novega vozlišča v gručo in proces uravnoteženja vsebnikov ter vpliv na povprečni odzivni čas storitve

Najbolj obetavna orodja oz. tehnologije in njihove rezultate meritev bomo po zmogljivosti primerjali s trenutnim sistemom, nameščenim v oblaku, kar bo osnova za odločitev o primernosti sistema vsebnikov za produkcijsko uporabo v podjetju.

Pri funkcionalnosti orodij nas bodo predvsem zanimali podpora za upravljanje s konfiguracijami, varnost, enostavnost uporabe, sledljivost spremembam, podpora za stalno integracijo, upravljanje z občutljivimi podatki in primernost programskih

vmesnikov, da lahko integriramo novo okolje v obstoječa orodja podjetja. Prav tako mora naša rešitev izpolnjevati, kolikor se dá, manifest, definiran za 12-faktorski razvoj aplikacij.

Za našo končno odločitev bo pomembna tudi skupnost okrog orodij in morebitna komercialna podpora. Pri tem kriteriju smo upoštevali predvsem formalno definiranost upravljanja produkta, aktivnost podpornih skupin, morebitno prosto dostopnost kode in to, ali imajo vzpostavljen sistem za prijavo.

2. Tehnologija vsebnikov in orodja za orkestracijo

Kljub temu, da je Docker približal vsebnike širši množici, je uporaba le-teh v produkciji še vedno relativno zapleteno opravilo [22], zato so podjetja, kot je npr. Google, potrebovala več generacij sistemov (*Babysitter*, *GlobalQorkQueue*, *Borg*, *Omega*) [23], da so prišla do primerne rešitve. Treba se je zavedati, da morajo produkcijska okolja zadostovati strogim varnostnim zahtevam, biti morajo visoko razpoložljiva in nazadnje tudi ekonomsko upravičena. V vozliščih, kjer hkrati poganjamo različne storitve, je treba upravljati s prioriteto zahtev in rešiti problem izolacije med aplikacijami, saj bi v nasprotnem primeru posamezni proces lahko zasedel vse sistemske vire in onesposobil preostale storitve na vozlišču [7]. Treba je vzpostaviti mehanizem, ki zna samodejno zaznati različne vrste okvar in samodejno odpraviti napake. Te zmogljivosti ne sodijo v osnovni nabor funkcionalnosti vsebnikov, zato je bilo treba razviti orodja za njihovo orkestracijo in razporejanje.

Orkestracija je proces, ki samodejno razporeja, koordinira in upravlja zapletene računalniške sisteme, storitve in vmesne sporočilne sisteme. Definirana je tudi kot proces, v katerem sistem samodejno in glede na zahteve operaterja najboljše razporedi naloge, da lahko glede na razporožljive vire najboljše poganja zastavljene naloge. Ob okvari posameznih vozlišč oz. drugih mrežnih ali strojnih okvarah orkestracijski sistem poskrbi, da samodejno zazna napako in glede na dane vire znova razporedi naloge, tako da delujejo, kolikor se dá optimalno.

Po pregledu trenutnih rešitev smo v ožji izbor umestili naslednja orkestracijska orodja: *Google Kubernetes*, *Apache Mesos* in *Docker Swarm*. Za ta orodja smo se odločili predvsem zaradi velike podpore v skupnosti in zaradi uspešnih primerov uporabe v produkciji večjih podjetij. *Kubernetes* se uporablja znotraj podjetja Google in je razporožljiv tudi kot storitev *Google Cloud*. *Docker Swarm* je uradno podprt s strani podjetja Docker in je osnova njihovega plačljivega produkta za orkestracijo vsebnikov. Za vključitev sistema *Apache Mesos* pa smo se odločili zato, ker je to uveljavljena rešitev za upravljanje strežnikov v podjetjih, kot sta Twitter in Apple. Več o lastnostih orodij bomo razložili v nadaljevanju.

Omenili bi še Amazonovo rešitev *EC2 Container Service* (*ECS*). Kljub temu, da orodje nudi podobno funkcionalnost kot *Kubernetes* in je zelo priljubljeno med uporabniki *AWS*, ni bilo izbrano kot primerna rešitev. Glavna razloga za to sta bila, da izvorna koda ni prostodostopna in da deluje le znotraj Amazonovega oblaka. Kljub temu, da smo eden večjih uporabnikov njihovega oblaka v Avstraliji, smo želeli neodvisno rešitev, ki bi nam omogočila morebitno selitev v *Microsoft Azure* ali *Google Cloud.*

Poleg zgoraj omejenih orodij obstajajo tudi druga, na primer Nomad, RancherOS, CoreOS, itd., ki imajo zelo dober potencial, vendar so premalo preizkušena in se veliko spreminjajo, tako da smo se odločili, da trenutno še niso resna alternativa.

V nadaljevanju magistrskega dela bomo sprva strnjeno opisali tehnologijo vsebnikov in implementacijo *Docker*, ki je osnovni in ključni gradnik naših orkestracijksih orodij. Nato bomo predstavili zgoraj omejena orkestracijska orodja in jih medsebojno primerjali po funkcionalnostih.

2.1 Tehnologija vsebnikov

Vsebniki so posebna vrsta virtualnih strojev, ki virtualizacije ne izvajajo na fizičnem nivoju, temveč na nivoju operacijskega sistema (ang. *Operating System-Level Virtualization*) [24]. Omogočajo virtualna okolja z ločenim pomnilnikom, centralno procesno enoto, omrežjem in diskovnim prostorom.

Začetki tehnologije vsebnikov segajo že v leto 1979 z uvedbo ukaza *chroot* v inačici *Unix* 7. Nato je leta 1998 FreeBSD izdal dopolnjeno verzijo jail. Sledili so še ostali proizvajalci, kot sta Solaris z *zones* [25] in IBM AIZ z *workload partition.*

Uradna podpora za vsebnike je bila v Linuxu jedru dodana leta 2007 z uvedbo podpore imenskih prostorov (ang. *namespaces*) in kontrolnih skupin (ang. *control groups*).

Imenski prostor omogoča izolacijo virov znotraj vsebnika, tako da ne more vplivati na preostale vsebnike in gostitelja samega. Možno je izolirati procese, tako da vsak imenski prostor vidi le svoje procese in podprocese. Na datotečnem sistemu lahko izoliramo vsebino na podmape gostiteljevega datotečnega sistema. Privzeto ima tudi vsak imenski prostor edinstveno ime vozlišča, domene in po potrebi svoje zaprto

omrežje. Številke procesov, procecorski (ang. proc) virtualni datotečni sistem in pomnilniški prostor so edinstveni, kar onemogoča zlorabo sistema.

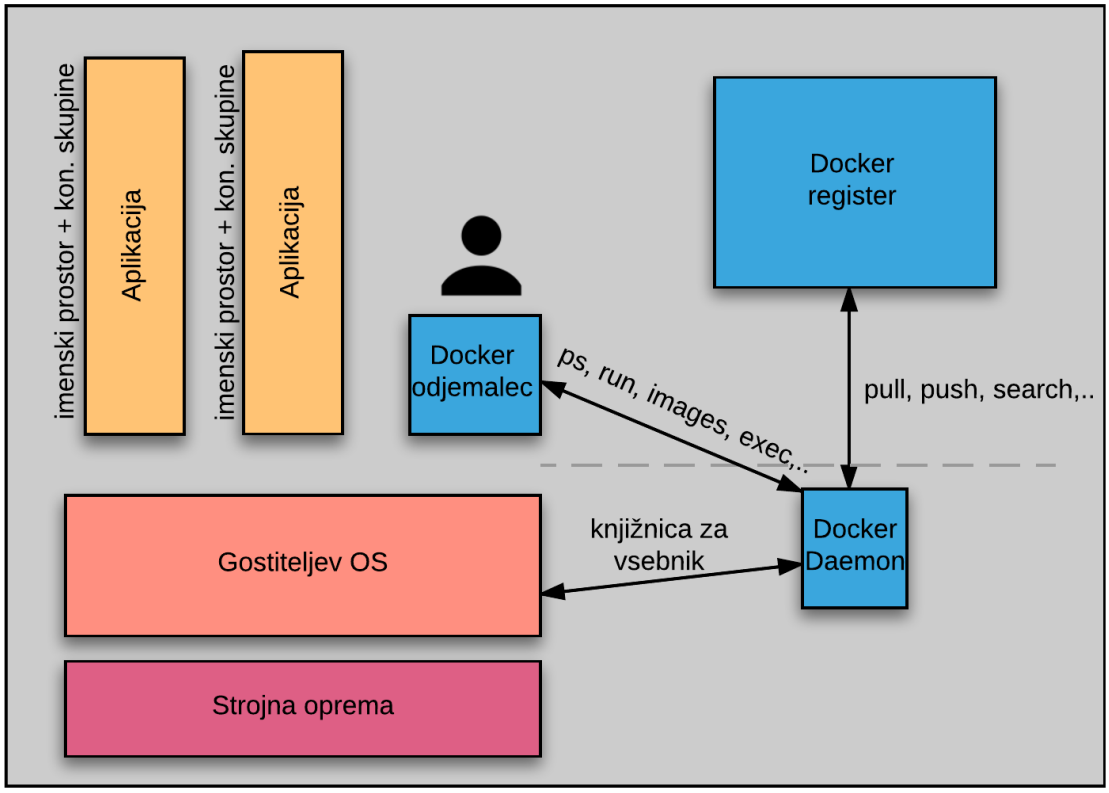
Kontrolne skupine omogočajo omejevanje največje količine dodeljenega spomina za posamezni vsebnik. Za omejevanje procesorske moči in vhodno-izhodnih naprav pa lahko določimo delež uporabe glede na vse razpoložljive vire. To je le omejtitev navzgor, saj bo v primeru prostih virov posamezni vsebnik lahko upobljal vse razporožljive vire. Za bolj napredno uporabo je mogoče omejiti tudi to, na katerih jedrih se izvajajo *CPU* operacije.

Kljub temu, da je bilo razvitih veliko rešitev za tehnologijo vsebnikov, kot so *OpenVZ*, *LXC*, *rkt,* je trenutno najbolj priljubljena rešitev *Docker*, ker je uporabniku najbolj prijazna.

V nadaljevanju bomo natančneje pregledali tehnologijo vsebnikov *Docker*, ki je tudi edina, ki je podprta na vseh preizkušenih in v nadaljevanju opisanih orkestracijskih orodjih.

2.1.1 Docker

Zgodnji uporabniki Dockerja so bili predvsem razvijalci, saj jim je le-ta omogočil enostavno izolacijo in ovijanje (enkapsulacijo) aplikacij znotraj vsebnikov ter hkrati enostavno vzpostavitev sistemskega okolja brez posredovanja sistemskih in mrežnih inženirjev [26].



Slika 2-1: Poenostavljena arhitektura *Docker* na enem vozlišču

Na sliki 2-1 je prikazana poenostavljena arhitektura Dockerja na enovozliščni namestitvi. *Docker* je sestavljen iz treh komponent in sicer odjemalca, storitve, ki jo bomo v nadaljevanju poimenovali *Docker*, in registra *Docker*. Za izvajanje ukazov potrebujemo odjemalec, ki uporablja programski vmesnik za interakcijo s storitvijo. Za tipično namestitev potrebujemo tudi Dockerjev register, kjer imamo shranjene privatne slike vsebnikov. Če uporabljamo prostodostopne vsebnike, zadostuje prostodostopni register *Docker Hub*.

Docker je mogoče poganjati na fizičnem ali virtualnem stroju. Povprečno lahko na enem strežniku poganjamo 40–50 vsebnikov, kar pa je predvsem odvisno od

potrebnih računalniških zahtev posamezne storitve. Tipični ukazi so prikazani na primeru 2-1.

```
# zažene interaktivni vsebnik z Linuxovo distrubucijo Ubuntu 14.04
$ docker run –it –name ubuntu ubuntu:14.04 /bin/bash

# zažene strežnik Nginx kot storitev v ozadju, izpostavljeno na vratih 8181
$ docker run –d nginx –name nginx –p 8181:80

# zaustavi vsebnik Nginx
$ docker stop nginx

# odstrani vsebnik Nginx
$ docker rmi nginx
```

Primer 2-1: Osnovni ukazi *Docker*

Poleg upravljanja življenjskega cikla vsebnikov *Docker* omogoča tudi kreiranje slik. Zato uporablja standarizirano datoteko, poimenovano *Dockerfile*, v kateri določimo zaporedje ukazov, ki opisuje našo sliko po meri.

```
FROM ubuntu:latest

CMD apt-get update
CMD apt-get install –y nginx
COPY static-html-directory /usr/share/nginx/html
EXPOSE 80 443
CMD [“nginx”,”-g”, “daemon off;”]
```

Primer 2-2: Primer datoteke *Dockerfile*

V primeru 2-2 je prikazan postopek kreiranja slike vsebnika, ki vsebuje spletni strežnik in stran po meri. Ti osnovni gradniki so zelo pomembni, saj so osnovni elementi stalne integracije in pogoj za 12-faktorsko metodologijo razvoja aplikacij.

Podobno kot preostale implementacije vsebnikov *Docker* uporablja jedrni funkciji kontrolne skupine (ang. *control group*) in imenskega prostora (ang. *namespace*), da zagotovi izolacijo in varnost virov, vendar je njegova uporaba zelo poenostavljena in je večina operacij skritih pred končnim uporabnikom, kot je razvidno iz primera 2-3.

```
# Primer zagona vsebnika s 512MB spomina in dovoljene do dvakrat več procesorske moči
$ docker run –memory=”512MB” --cpu-shares=2048
```

Primer 2-3: Omejevanje virov vsebnika

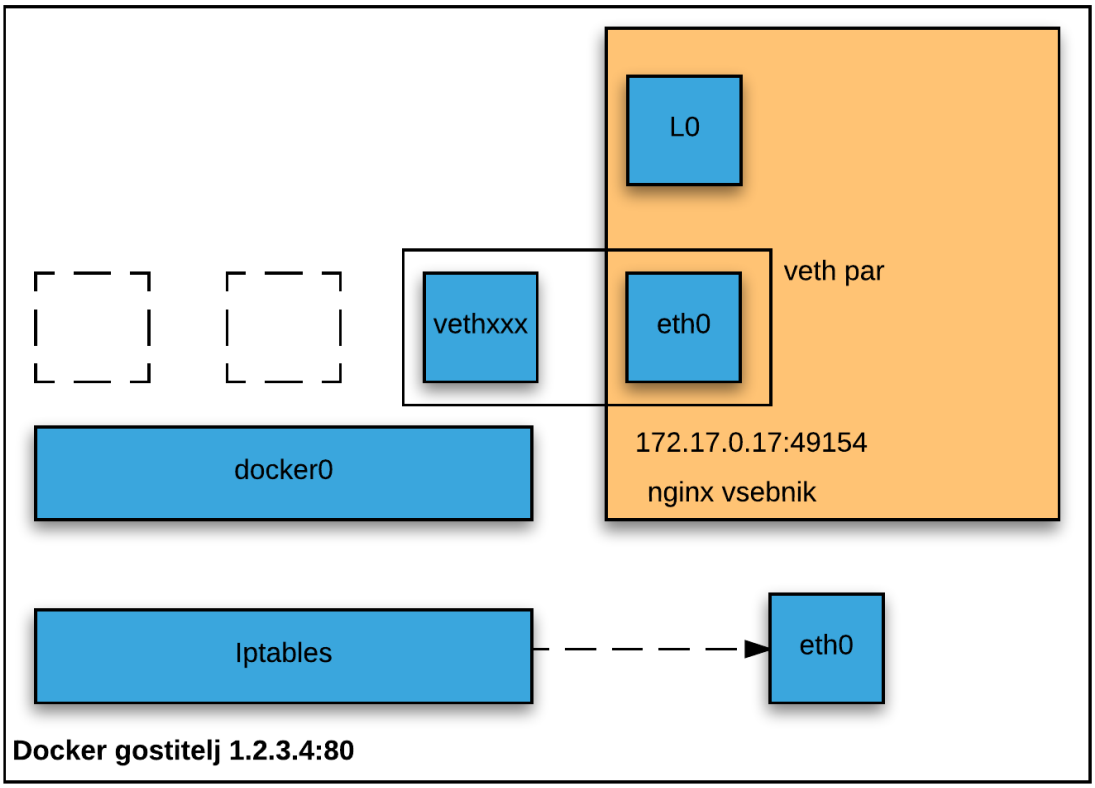
2.1.1.1 Omrežna povezanost

Način, kako lahko vsebniki medsebojno komunicirajo, je zelo pomemben, saj je to ključni element varnosti in predpogoj za večvozliščne namestitve vsebnikov in s tem varna produkcijska okolja.

Trenutno so podprti naslednji načini, in sicer:

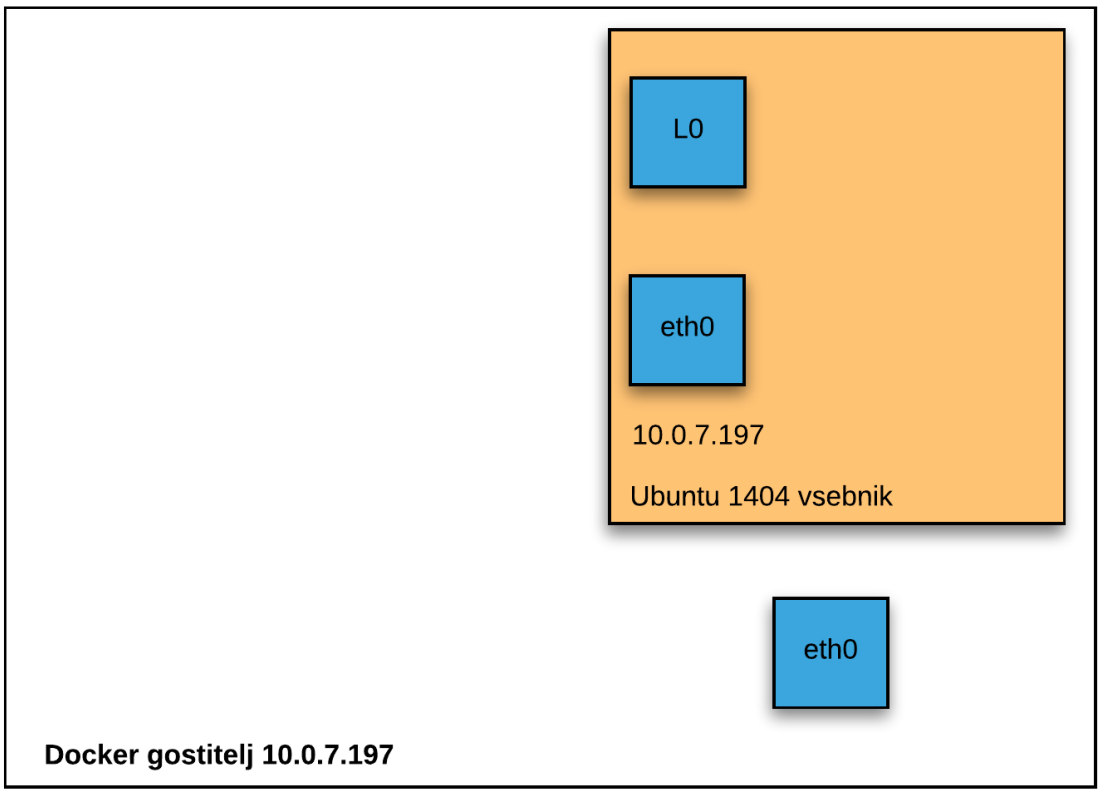
- način mosta (ang. *bridge mode*),
- način gostitelja (ang. *host mode*),
- omrežje z načinom vsebnika (ang. *container mode networking*),
- brez omrežja.

Način mosta je privzeti način, s katerim *Docker* vzpostavi virtualno omrežno napravo *docker0* in samodejno preusmerja vse pakete med omrežnimi napravami. Privzeto lahko vsi vsebniki na gostitelju komunicirajo med sabo posredno prek interno dodeljenih *IP*-naslovov. V tem načinu je mogoče tudi objaviti posamezna vrata na vsebnikih, kar omogoča, da zunanji uporabniki, prek gostiteljevega *IP*-naslova, dostopajo do storitve. Način mosta je prikazan na sliki 2-2.



Slika 2-2: Omrežna povezanost *Docker* na način mosta

V omrežnem načinu gostitelja, prikazanem na sliki 2-3, je izolacija med vsebniki izklopljena in je omrežna povezava neposredno izpostavljena vsebnikom. To v praksi pomeni, da imajo vsi vsebniki dodeljen gostiteljev *IP*, kar posledično pomeni, da so zunanjim odjemalcem storitve prosto dostopne. Hkrati je treba biti tudi pozoren na zasedenost vrat, saj je samodejno dodeljevanje nezasedenih vrat na gostitelju izklopljeno. Takšen način delovanja omrežja je hitrejši, a bolj izpostavljen nevarnostim.



Slika 2-3: Gostiteljev način omrežja *Docker*

Omrežje z načinom vsebnika pa omogoča, da si dva ali več vsebnikov deli enak omrežni prostor. V tem primeru gre za inačico načina mosta z enakimi lastnostmi, le da si vsebniki delijo skupni *IP*-naslov. Takšen hibridni način je zelo zanimiv predvsem za večvozliščna okolja, kar je tudi razvidno iz nekaterih rešitev.

2.2 Kubernetes

Kubernetes je odprtokodno orkestracijsko orodje, razvito s strani podjetja Google Inc., ki s pomočjo vsebnikov omogoča upravljanje z aplikacijami med različnimi vozlišči, nudi pa tudi preproste mehanizme stalne namestitve in horizontalno razširljivost aplikacij.

Orodje je rezultat več kot 10 let izkušenj Googla, kako uporabljati vsebnike v produkciji s 1000 in več vozlišči. Orodje je zasnovano na podlagi predhodnikov Kubernetesa, sistemov *Borg* in *Omega*, ter na izkušnjah, nabranih z uporabo teh

orodij v produkciji. Zanimanje v skupnosti za to rešitev je veliko, kar dokazuje tudi dejstvo, da je to eden najbolj priljubljenih projektov na spletnem servisu *GitHub*.

Kubernetes obljublja naslednje prednosti:

- **Enostavnost:** preproste komponente, lahko za namestitev in prosto dostopno.
- **Prenosnost:** sistem je mogoče namestiti v oblaku, na fizičnih strežnikih ali v hibridnih oblakih.
- **Razširljivost:** zaradi modularne zasnove in komponentne sestave je zelo enostavno razširiti funkcionalnost glede na specifične potrebe.
- **Samozdravljenje (ang. *self-healing*):** vgrajeni sistemi omogočajo samodejni zagon aplikacij ob izpadu, samodejno umestitev na vozlišča glede na zasedenost in samoreplikacijo ob povečanem prometu.

2.2.1 Komponente

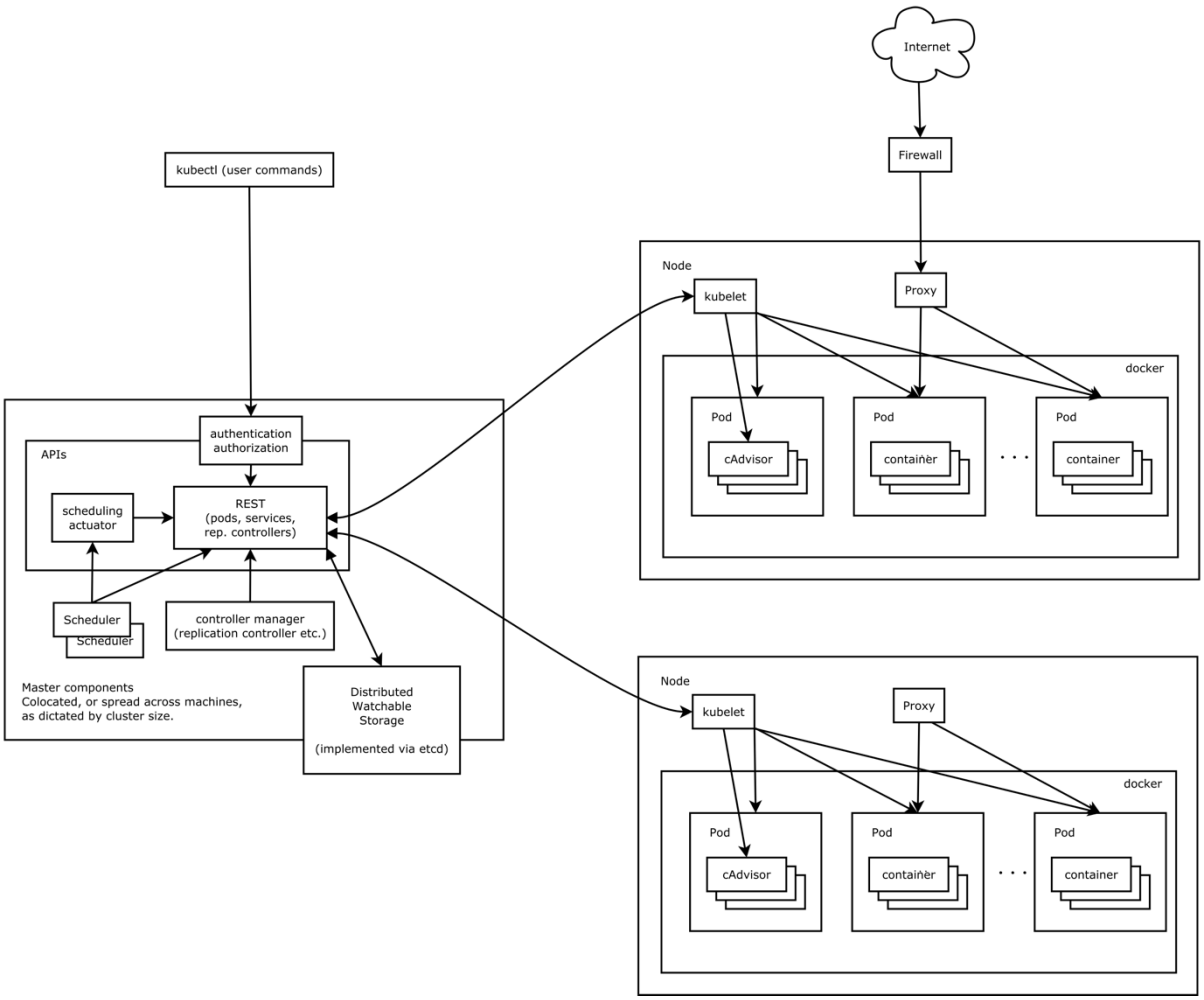
Kubernetes je sestavljen iz naslednjih komponent:

- **Vozlišče:** je fizični ali virtualni strežnik, ki poganja *Kubernetes Daemon*, ki upravlja s stroke (ang. *pod*).
- **Stroki (ang. *pods*):** so skupina, med sabo logično povezanih vsebnikov s skupnim diskovnim prostorom. To je najmanjša enota, ki jo je mogoče kreirati, razporejati in upravljati s pomočjo *Kubernetes*. Strok (ang. *pod*) lahko kreiramo posamezno, vendar je priporočljivo, da jih urejamo s pomočjo replikacijskih nadzornikov ali z namestitvenimi načrti.
- **Replikacijski nadzornik:** upravlja z življenjskim ciklom stroka (ang. *pod*) in zagotavlja, da imamo v našem okolju *Kubernetes* točno določeno število repliciranih podov med vozlišči.
- **Namestitveni načrt:** omogoča deklarativno nadgradnjo replikacijskega nadzornika ali strokov (ang. *pods*). Podobno kot replikacijski nadzornik zagotovi, da imamo na dani gruči zahtevano število strokov (ang. *pods*), vendar omogoča tudi samodejno nadgradnjo in mehanizme razveljavljanja (ang. *rollback*).
- **Storitev:** omogoča izpostavitev skupine strokov na podlagi etiket kot stalno tekočo storitev. Zagotavlja abstrakcijo storitve, ki ne glede na število strokov in spremembe vozlišč ostane nespremenjena.
- **Etikete:** so pomembni sestavni element gruč *Kubernetes*. S pomočjo etiket lahko označimo stroke, replikacijske nadozrnike itd., kar omogoča organizacijo virov *Kubernetes* v organizacijske strukture.

- **Označevanje (ang. *annotation*):** so metapodatki po meri, ki jih lahko po meri dodajamo na sestavne elemente *Kubernetes*. Po navadi jih uporabljamo kot identifikator za zunanja orodja.
- **Ključ-vrednost CoreOS etcd:** uporabljen je za shranjevanje metapodatkov o vozliščih in trenutnih stortivah, namestitvah itd.

2.2.2 Arhitektura gruče

Gručo *Kubernetes* sestavljajo delovna vozlišča (ang. *kubelet*) in kontrolno vozlišče, ki vsebuje komponente razporejevalnika, programskega vmesnika itd. (prikazano na sliki 2-4). Za koordinacijo in izmenjavo podatkov med različnimi vozlišči skrbi *CoreOS etcd*, distribuirani podatkovni sistem.



Slika 2-4: Arhitektura Kubernetes

Bistvena arhitekturna razlika orodja *Kubernetes* v primerjavi s preostalimi orodji je, da ima vsak strok dodeljen svoj interni *IP*-naslov. To so dosegli tako, da privzeto uporabljajo vsebnikov mrežni način. Prednost take mrežne ureditve je, da ima lahko vsak strok vnaprej dodeljena mrežna vrata, tako da ni potrebno preslikovanje vrat glede na vozlišče, prav tako lahko vsak strok komunicira s preostalimi stroki, tudi če se nahaja na drugem vozlišču. Arhitekturno so stroki zelo podobni klasičnim virtualnih strojem, kar poenostavi selitev na vsebnike.

Delovno vozlišče *Kubernetes*
Delovno vozlišče vsebuje vse potrebne storitve, da lahko uspešno poganja aplikacije in izvaja naloge, dodeljene s strani kontrolnega vozlišča.

- Komponenta *kubelet* upravlja z vsebniki, stroki, slikami in s skupnim diskovnim poljem.
- *Kube-proxy* omogoča enostaven vmesnik in razporejanje prometa (ang. *load balancing*), ki ga uporablja vir storitev za razporejanje zahtev med stroki.

Če je kontrolno vozlišče nedosegljivo, delovna vozlišča še vedno nemoteno delujejo, vendar je prerazporejanje in dodajanje novih opravil nemogoče.

Kontrolno vozlišče
Kontrolno vozlišče so možgani gruče *Kubernetes*, saj razporejajo in nadzorujejo vse storitve. Vozlišče je sestavljeno iz različnih komponent, in sicer:

- **Razporejevalnik** ureja nerazporejene strokov na vozlišča prek programskega vmesnika /*binding*. Razporejevalnik je izmenljiv, kar omogoča rešitve po meri.
- **Strežnik *Kubernetes API*** je komponenta, ki omogoča interakcijo z gručo prek programskih vmesnikov. Zasnovan je prek vmesnika *REST* in omogoča operacije *CRUD*, ki jih validira in ob avtoriziranem klicu shrani spremembe v ditribuirani sistem *ETCD*.
- ***CoreOS etcd*** je distribuirani sistem za shranjevanje podatkov v obliki ključa - vrednosti, ki služi za shranjevanje stanja gruče. Za shranjevanje in usklajevanje podatkov *ETCD* uporabljajo soglasni *protocol raft* [27], ki je zelo priljubljen v sodobnih orkestracijskih sistemih.

2.3 *Apache Mesos*

Apache Mesos je prav tako odprtokodni projekt za upravljanje distribuiranih sistemov, razvit pod okriljem fundacije Apache Inc. V primerjavi s preostalima orodjema gre za relativno starejši projekt, ki pa je podporo za vsebnike dodal naknadno. Začetki orodja segajo na univerzo Berkeley, kjer so želeli razviti operacijski sistem za podatkovni center, kjer bi bilo mogoče upravljati z več strežniki kot celoto.

Apache Mesos avtomatizira upravljanje z računalniškimi viri več strežnikov in razporejanje procesov med strežniki, poenostavi komunikacijo med procesi in nudi enostaven grafični vmesnik ter programske vmesnike za nadziranje in upravljanje s storitvami. Cilj produkta je, da končnemu uporabniku predstavi gručo strežnikov kot celoto, tako da se lahko razvijalci osredotočijo na rešitve in produkte brez potrebe, da bi se ukvarjali s kompleksnostjo distribuiranih sistemov in upravljanjem le-teh.

Veliko časa so posvetili enovitemu programskemu in grafičnemu vmesniku, v katerem je celoten podatkovni center predstavljen kot celota. S tem so želeli omogočiti primerno ogrodje, kjer je enostavno razviti skripte po meri za specifične zahteve.

Za namene magistrskega dela bomo uporabili implementacijo *DC/OS*, ki temelji na jedru *Apache Mesos* in vključuje skupek orodij in storitve za poenostavljeno uporabo gruče.

2.3.1 Komponente

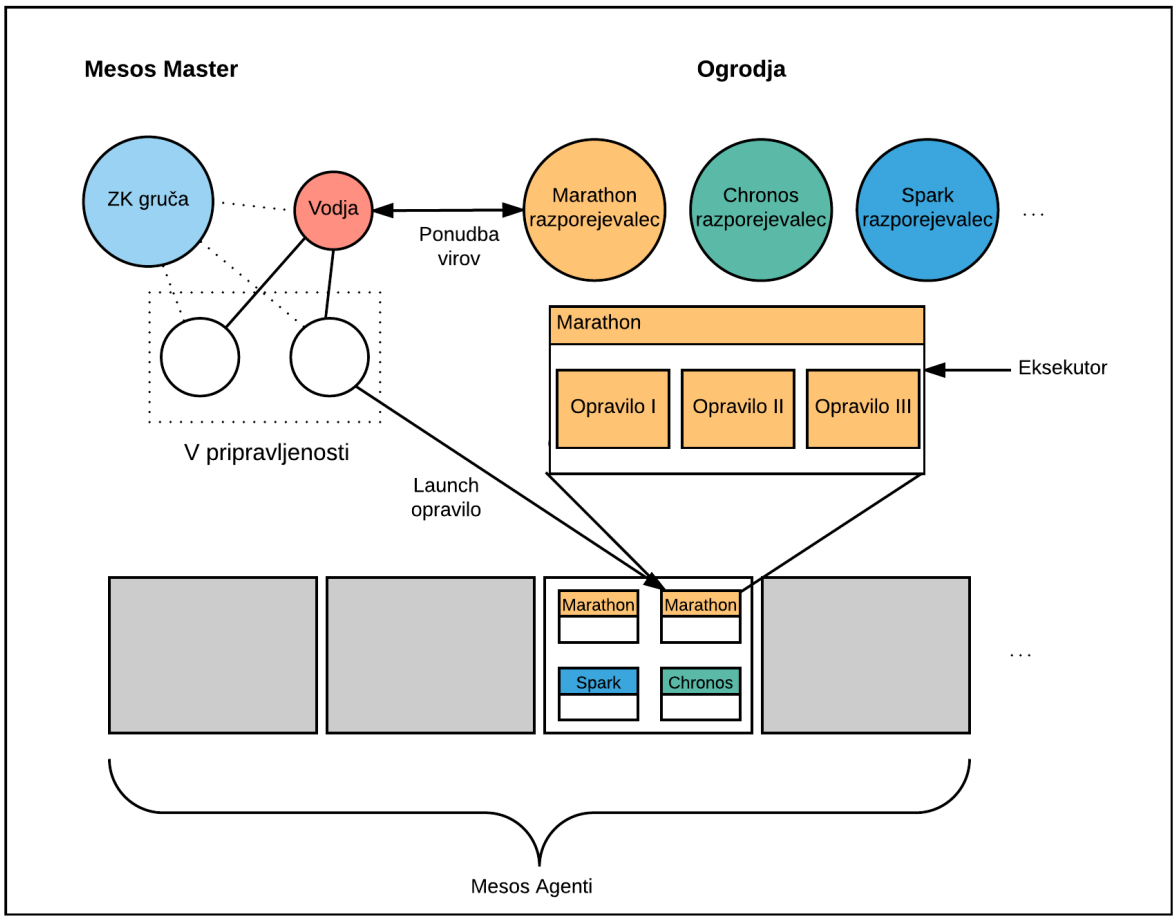
Apache Mesos je sestavljen iz naslednjih komponent:

- ***Master*** agregira računalniške vire, ki jih nudijo vsi odjemalci in jih nudi/razporeja registriranim ogrodjem (ang. *frameworks*). Poročilo o uporabi dobi s strani odjemalcev in razporeja vire registriranim storitvam, kot sta npr. *Marathon* ali *Spark*. Arhitektura je zelo podobna kot pri *Docker Swarm* (več o tem v nadaljevanju).
- **Odjemalec** izvaja diskretno nalogo *Mesos* na zahtevo ogrodja. V posamezni literaturi so odjemalci opisani tudi kot agenti *Mesos* ali delovna vozlišča.
- **Razporejevalec (ang. *scheduler*)** razporeja storitve med odjemalci.

- **Uporabnik oz. odjemalec** je zunanja ali interna aplikacija gruče, ki sproži proces (npr. uporabnik, ki izvede zahtevo za aplikacijo *Marathon*).
- **Naloga** je enota dela v ogrodju *Mesos*, ki je izvedena na odjemalcu *Mesos*.
- **Izvajalec (ang. *executor*)** zažene in upravlja naloge na odjemalcih.

2.3.2 Arhitektura

Gručni upravljalec *Apache Mesos* s kontrolerjem *master* upravlja in komunicira z odjemalci na vozlišču. Vozlišča poganjajo program *Mesos* v načinu odjemalca, ki sprejema naloge s strani kontrolerja *master*. Tudi v tem primeru gre za podobno arhitekturo kontroler/odjemalec, kot jo vidimo pri *Kubernetes* in Swarmu. Prav tako kot *Apache Mesos* uporablja ključ/vrednost za shranjevanje metapodatkov in trenutnega stanja sistema tudi distribuirani sistem *ZooKeeper*.

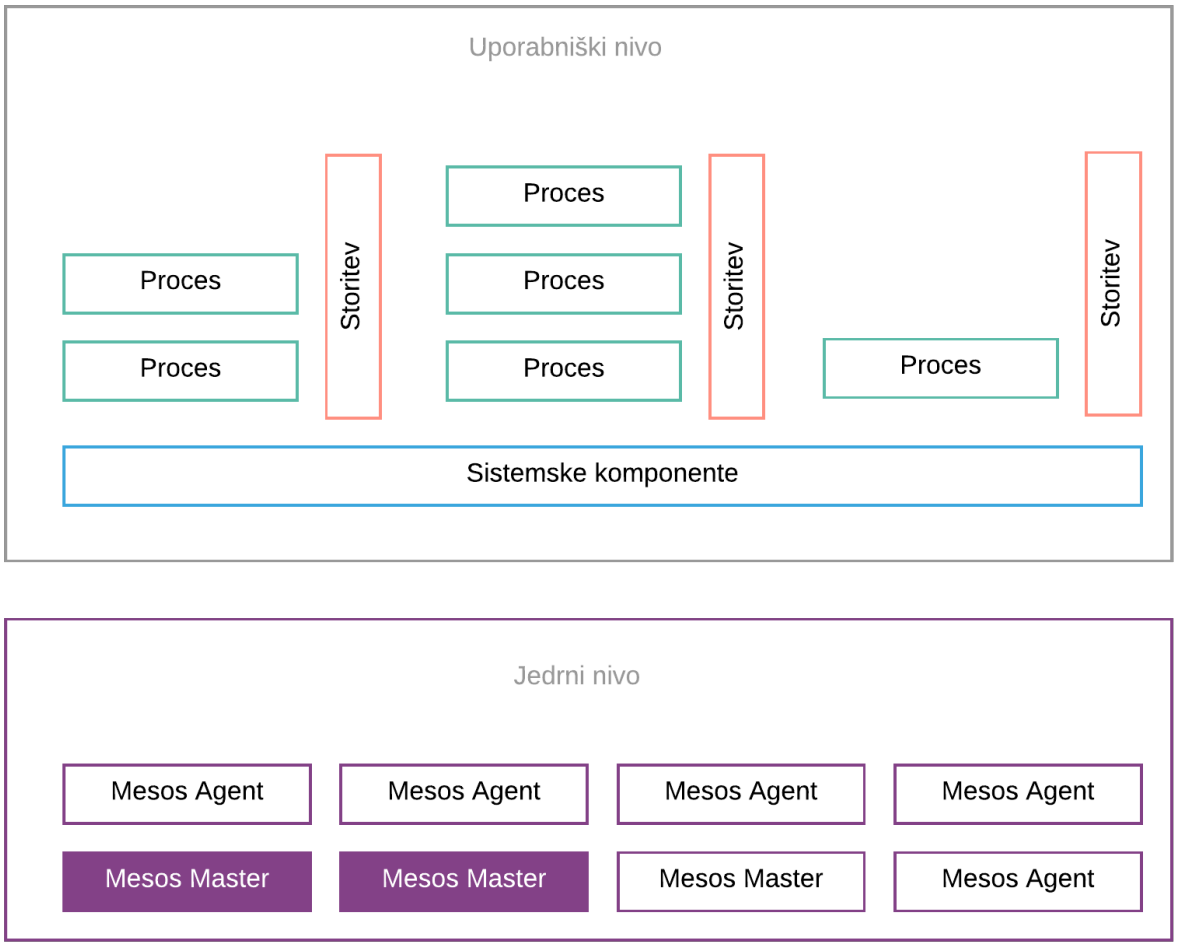


Slika 2-5: Osnovna arhitektura *Mesos*

Kot je razvidno iz slike 2-5 *Mesos* privzeto podpira različna ogrodja/razporejevalce, kot so *Spark* (ogrodje za obdelavo podatkov v relnem času), *Hadoop* (orodje za

obdelavo velikih količin podatkov) in *Marathon* (orodje za orkestracijo vsebnikov). V našem primeru bomo uporabljali le *Marathon*, saj nas primarno zanimajo dolgoročne tekoče storitve z vsebniki, vendar bomo pri končni odločitvi upoštevali vse funkcionalnosti tega orodja.

Arhitektura *Mesos* je ločena na dva nivoja, in sicer na jedrni in uporabniški nivo, kot je prikazano na sliki 2-6.



Slika 2-6: Poenostavljen diagram *DC/OS Apache Mesos*

Na jedrnem nivoju *Mesos DC/OS* upravlja z dodelitvijo virov in razporejanjem nalog v gruči. Na tem nivoju imamo dva procesa, in sicer odjemalec *Mesos* in *master*, ki smo ju opisali v prejšnjem poglavju.

Na uporabniškem nivoju imamo sistemske komponente in storitve, kot so npr. *Chronos* (distribuirani sistem za izvajanje nalog) ali *Kafka* (orodje za izmenjavo sporočil med sistemi).

Sistemske komponente so samodejno nameščene in privzeto aktivne:

- **Operaterski usmerjevalnik** je odprtokodni strežnik *NGINX* po meri, ki omogoča centralno avtentikacijo in vrata do storitev *DC/OC*.
- ***ZooKeeper*** je distribuirani sistem za shranjevanje ključev/vrednosti podatkov.
- ***Exibitor*** med namestitvijo samodejno namesti *ZooKeeper* in nudi poenostavljen grafični vmesnik za upravljanje le-tega.
- ***Mesos-DNS*** s pomočjo protokola *DNS* nudi odkrivanje storitev, kar omogoča tradicionalnim aplikacijam enostaven način povezovanja različnih storitev.
- **Distribuirani posrednik *DNS*** (ang. *proxy*) se uporablja za interno razporejanje zapisov *DNS*.
- ***Marathon*** je izvirni način za izvajanje in nadziranje storitev z vsebniki, enakovreden sistemu *Init* v okoljih *Unix*.

Poleg sistemskih komponent je mogoče izvajati tudi uporabniške storitve, ki jih definiramo kot naloge, ki jih razporejevalec dodeli različnim odjemalcem glede na razpoložljivost virov. Primer take storitve je, denimo, strežnik *Apache*, ki ga lahko zaženemo s pomočjo Marathona.

2.4 *Docker Swarm*

Docker Swarm je orkestracijska rešitev podjetja Docker Inc. in je postal skupni element vsebnika *Docker* od različice 1.12. Rešitev je relativno nova, vendar ima veliko podpore v podjetju, saj je temelj njihove plačljive platforme.

Podobno kot preostali sorodni produkti skuša *Docker Swarm* ponuditi rešitev, ki omogoča uporabo vsebnikov za produkcijska okolja, kjer so visoka rezpoložljivost, stalna integracija in samodejno obnavljanje storitev mogoči.

Produkt je relativno nov in zato se vmesniki in funkcionalnosti veliko spreminjajo. Za naše okolje smo testirali novo generacijo načina *Swarm*, ki je neposredno integriran z novo različico vsebnika *Docker* in se bistveno razlikuje od prejšnje inačice.

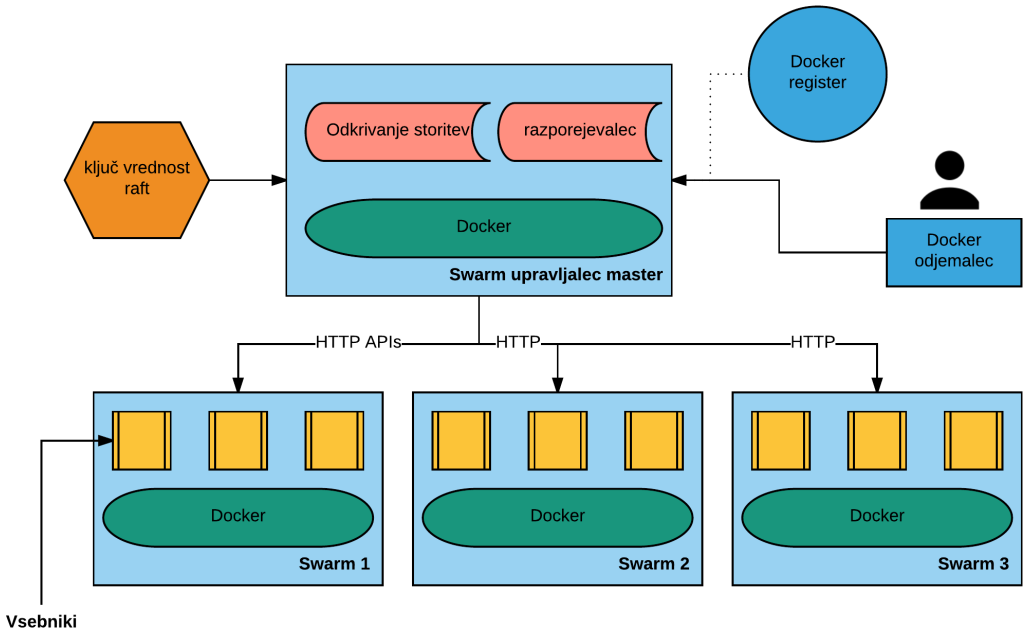
Bistvena razlika nove inačice Swarma je dodaten tip objekta, in sicer storitev – gre za kombinacijo storitve *Kubernetes* in namestitvenega načrta, s pomočjo katerih lahko

definiramo število kopij vsebnika, razporejenega v gruči, in opcijsko določimo vrata, prek katerih bo storitev dosegljiva.

Prav tako moramo biti pozorni, da ga ne zamešamo z *Docker SwarmKit*, ki je neodvisna komponenta, namenjena razvoju orkestracijskih in distribuiranih sistemov po meri.

2.4.1 Arhitektura

Podobno kot preostali rešitvi arhitekturo *Docker Swarm* sestavljajo kontrolno vozlišče *Swarm Master* in delovna vozlišča (slika 2-7). *Swarm* v primerjavi s *Kubernetes* podpira tudi visoko razpoložljivo kontrolno vozlišče, ki je lahko definirano z dvema ali več pomožnimi vozlišči, ki ob izpadu glavnega lahko prevzamejo aktivno vlogo.



Slika 2-7: Tipična arhitektura *Docker Swarm*

Podobno kot preostala orodja tudi *Swarm* uporablja distribuirani sistem ključ-vrednost (ang. *key/value system*) za shranjevanje metapodatkov o virtualnih omrežjih in stanju opravil/storitev. V primerjavi s preostalima sistemoma *Swarm* z novo različico ni več odvisen od zunanjega sistema, ampak sam implementira protokol *raft* kot del zagona

Swarm. To zelo poenostavi upravljanje in postavitev okolja, saj ni več treba postavljati in vzdrževati samostojnega sistema ključ-vrednost. Sistem je tudi visoko razporožljiv, saj hrani vsako kontrolno vozlišče metapodatke celotne gruče.

V primerjavi z drugimi podobnimi orodji *Swam* uporablja standardni programski vmesnik *Docker*, kar pomeni, da je za končnega uporabnika upravljanje z gručo zelo podobno kot uporaba Dockerja samega. V praksi to pomeni, da so orodja, kot so *Dokku*, *Docker Machine* in *Jenkins* popolnoma podprta v sistemu. Razvoj podpore za *Docker composer* še poteka, saj je rešitev relativno nova, ampak pričakovati je, da bo to implementirano z naslednjo različico.

Ena najpomembnejšeh sprememb z novo različico je uvedba podpore za usmerjanje *mesh*, ki je osnova za samodejno razporejanje prometa za storitev *Swarm*. Nov algoritem omogoča, da je storitev dosegljiva na enakih vratih na vseh vozliščih gruče, tudi v primeru, če storitev ni nameščena. V slednjem primeru bo *Swarm* poskrbel, da bo interno preusmeril promet na vozlišče, kjer je storitev dejansko nameščena. To zelo poenostavi vzdrževanje in nameščanje dolgoročnih tekočih storitev v produkciji. Pred tem so bili potrebni posebni vsebniki, ki so s pomočjo programskega vmesika ažurirali seznam veljavnih vozlišč. Poleg tega, da je bil proces zamuden in slabo pregleden, je bilo treba tudi primerno zagotoviti visoko razporožljivost še ene dodatne storitve.

2.5 Primerjava orodij po funkcionalnosti

V tabeli 2.1 so strnjeno opisane lastnosti Kubernetesa, *Apache Mesos* in *Docker Swarm*. Z zeleno barvo so označene po našem mnenju najboljše rešitve. Spodaj pa sledi še opis lastnosti orodij, ki so najpomembejša za tekoče dolgoročne storitve.

	Kubernetes	***Mesos***	***Swarm***
Tip obremenitev	Tekoče dolgoročne storitve z vsebniki.	Tekoče dolgoročne storitve z vsebniki in preostala ogrodja, kot so *Hadoop*, Spark, itd	Tekoče dolgoročne storitve z vsebniki.
Definicija aplikacije	Napredni gradniški jezik, definiran s pomočjo strokov, namestitvenih načrtov, storitvev, skivnostmi, itd. Vsebniki znotraj enega stroka bodo pognani na istem gostitelju.	Napredni gradniki, definirani s pomočjo aplikacijske definicije. Kolokacija skupine vsebnikov na enem strežniku ni mogoča. Gradnik, podoben stroku, načrtovan v prihodnjih inačicah.	Primitivni gradniki storitev. Ni mogoče zagotoviti, da je skupina vsebnikov nameščena na enakem gostitelju. Ni še podpore za definicijo storitve več vsebnikov s pomočjo urejevalnika (ang. composer) Docker.
Razporejevanje prometa	Stroki so izpostavljeni preko storitve, s pomočjo katere imajo tudi podporo za samodejno razporejanje prometa s pomočjo razporejevalca prometa *AWS ELB* ali *Google Cloud.*	Aplikacije so lahko dostopne prek Mesos DNS, vendar ni neposredne podpore za samodejne razporejevalce prometa, kot je *AWS ELB*.	Razporejanje prometa poteka neposredno prek vrat, vendar ni vmesnega nivoja, ki objavi storitev neposredno prek razporejevalca prometa.
Shranjevanje metapodatkov gruče	*CoreOS etcd*	*Apache ZooKeeper*	Interna implementacija protokola *rafl*
Dnevniški zapisi in nadziranje (ang. *monitoring*)	Preverjanje delovanja aplikacije na dva načina, in sicer dosegljivost (ang. *liveness*) (če je aplikacija pognana) in pripravljenost (ang. *readiness*) (če je aplikacija pripravljena). Dnevniški zapisi so	Dnevniške zapise možno shraniti v *ELK*. Nadziranje možno le prek zunanjih orodij.	Dnevniški zapisi in nadziranje možni le s pomočjo zunanjih orodij.

	shranjeni v storitvi *ELK* (ang. *elasticsearch logstash kibana*). Dodatna nadziranja in shranjevanje dnevniških zapisov možna s pomočjo podprtih razširitev		
Skalabilnost in *performance*	1.000 vozlišč	50.000 vozlišč	Ni podatkov
Podprti vsebniki	*Docker* in *rkt*	*Docker* in *LXC*	*Docker*

Tabela 2-1: Primerjava Kubernetes, Mesos in Swarm po lastnostih

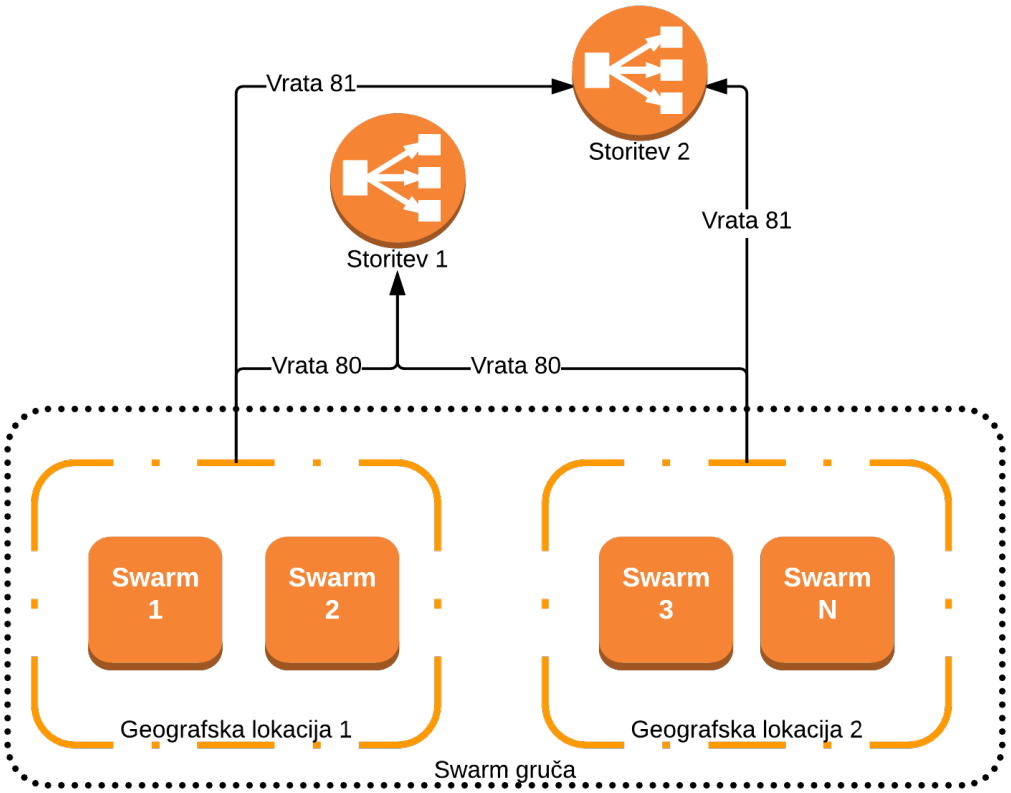
V primerjavi s preostalima orodjema je *Docker Swarm* zelo enostaven za namestitev in upravljanje. Z zadnjo različico ima vgrajen sistem ključ-vrednost, kar znatno zmanjša režijske stroške upravljanja infrastrukture. Upravljanje z vsebniki v gruči je zelo podobno kot upravljanje posameznega vozlišča, zato je rešitev zelo priljubljena med razvijalci. Prav tako z vgrajeno podporo za pokrivno omrežje (ang. *overlay network*) omogoča enostavno kreacijo povezanih omrežij med več vozlišči. Z novim gradnikom storitev, ki je bil dodan z zadnjo različico *Docker*, je zelo enostavno pognati visoko razpoložljive storitve (gl. primer 2-4).

```
# definira omrežje Webnet:
$ docker network create --overlay webnet

# zažene strežnik Redis z eno kopijo vsebnika:
$ docker service create --name redisdb --network webnet --replicas 1 redis:alpine

# zažene testno aplikacijo Hello, ki je dosegljiva na vseh vozliščih prek vrat 80
$ docker service create --name fri-hello --network webnet --publish 80:8181 --replicas 3 golja/echo-server
```

Primer 2-4: Zagon visoko razpoložljive storitve z *Docker Swarm*

Vendar ima enostavnost uporabe tudi svojo ceno, saj hitro nastanejo težave pri bolj zapletenih scenarijih. Primer je prav nov gradnik storitve *Swarm*. Težava je, da so dodeljena vrata edinstvena glede na gručo. To pomeni naslednje: Če imamo dve storitvi, ki morata delovati na vratih 80, moramo dodati še en vmesni razporejevalec prometa, ki deluje na standardnih vratih 80 in preusmeri promet na nestandardna vrata. Kot je prikazano na sliki 2-8, bi v našem primeru morali vzpostaviti *AWS ELB*

ali visoko razpoložljiv posredniški strežnik *haproxy*. V nasprotnem primeru bi zunanji uporabniki imeli težave, dostopati do vseh storitev, ki poslušajo na nestandardnih vratih.



Slika 2-8: Rešitev *Swarm* v primeru storitev, ki si delijo enaka vrata

Kljub temu, da je to mogoče vzpostaviti za produkcijska okolja, bi potrebovali orodje po meri, ki neposredno uporablja programske vmesnike Swarma. Naslednja pomanjkljivost gruče *Swarm* je pomanjkanje avtentikacije in avtorizacije. Trenutno ni nikakršnega mehanizma, na podlagi katerega bi lahko omejili operacije na gruči, kar za produkcijska okolja ni sprejemljivo. Opaziti je tudi, da je orodje še v aktivnem razvoju, kar vpliva na stabilnost produkta. Samo v zadnjem letu so izdali več kot deset različic z bistvenimi spremembami, kar povzroča veliko preglavic vzdrževalcem sistemov. Tolikšnim spremembam je težko slediti in jih je težko finančno upravičiti, saj bi bilo treba z vsako izdajo vsa trenutna okolja drastično spremeniti.

V nasprotju z *Docker Swarm* je *Apache Mesos* veliko bolj zrel produkt, saj obstaja na trgu že dalj časa. Podporo za vsebnike so dodali naknadno s komponento *Marathon*. Orodje se tudi že dalj časa uporablja v večjih produkcijskih okoljih, kot so Twitter,

Linkedin, BBC, Apple itd. Večinoma se uporablja za bolj tradicionalna okolja za obdelavo večje količine podatkov, v sistemih, kot so *Hadoop*, *Spark*, *Kafka* itd. Sistem privzeto že podpira napredne mehanizme za preverjanje stanja aplikacije, labele, omejitve na osnovi vozlišč itd. Kljub temu, da je funkcionalnost podobna preostalima orkestracijskima orodjema, se konceptualno razlikuje, saj "združi" več strežnikov v eno celoto. V preostalih orodjih, ki smo jih preučili, so komponente veliko bolj zamenljive in nepovezane. Prav zaradi tega je kompleksnost vzdrževanja take gruče bolj zahtevna in toga. Na splošno so režijski stroški upravljanja komponente *Mesos* zelo veliki, prav zaradi zapletenosti okolja. Pri arhitekturi se pozna, da je bila zasnovana v obdobju pred storitvami kot servis in nadvlada oblaka, saj je arhitektura primerna bolj za tradicionalne podatkovne centre. Zaradi fleksibilnosti orodja je mogoče na veliko načinov rešiti enak problem, kar je včasih dobro, vendar je zaradi tega zelo težko ugotoviti, katera je dejansko najboljša rešitev. Posledica navedenega je, da je v skupnosti veliko napačnih rešitev: npr. samo za mehanizme odkrivanja storitev je mogoče uporabiti *MesosDNS*, *Marathon-loadbalancer*, razporejanje prometa z virtualnimi *IP*-ji itd. Žal pa nobena storitev ne zna neposredno uporabljati programskih vmesnikov oblačnih ponudnikov, tako da je podobno kot pri Swarmu še vedno treba razvijati spremljevalna orodja, da je okolje primerno za produkcijsko uporabo v *AWS*.

Kubernetes je v primerjavi z *Mesos* veliko bolj preprosto orodje, ki je specializirano samo za orkestracijo in razporejanje vsebnikov. V primerajavi s Swarmom ima dodatne gradnike, ki omogočajo načrtovanje naprednejših tekočih stalnih storitev. To so dosegli tako, da so definirali svoj jezik *DSL*, ki na višji stopnji določi manjkajoče elemente, potrebne za visoko razporožljiva okolja. Elementi so veliko bolj preprosti in neodvisni – npr., če želimo pri Kubernetesu definirati visoko razpoložljivo storitev, moramo najprej določiti namestitveni načrt, ki določi sliko vsebnikov, število redundančnih kopij storitve, konfiguracijske parametre in labele, šele nato je mogoče definirati storitev, ki na osnovi prej definiranih label izbere, kateri vsebniki dejansko procesirajo zahteve.

```
---
apiVersion: "v1"
kind: "Service"
metadata:
  name: "prometheus"
  annotations:
    # this enables internal ELB
    service.beta.kubernetes.io/aws-load-balancer-internal: "0.0.0.0/0"
    domainName: "example-prometheus-development-v1"
service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:ap-southeast-
2:XXXXXXXXX:certificate/XXXXXXX
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
    service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
spec:
  ports:
    - port: 443
      targetPort: "http"
      name: https
  selector:
    service: "prometheus"
    role: "server"
  type: "LoadBalancer"
```

Primer 2-5: Definicija storitve *Kubernetes* z elementi *AWS*

```
---
apiVersion: "extensions/v1beta1"
kind: "Deployment"
metadata:
  name: "prometheus-server"
spec:
  replicas: 1
  template:
    metadata:
      labels:
        service: "prometheus"
        role: "server"
    spec:
      serviceAccount: prometheus
      serviceAccountName: prometheus
      containers:
      - name: "prometheus"
        image: "prom/prometheus"
        imagePullPolicy: Always
        ports:
        - containerPort: 9090
          name: "http"
```

Primer 2-6: Definicija namestitvenega načrta *Kubernetes*

Kot je razvidno iz primerov 2-5 in 2-6, lahko na preprost način definiramo storitev, ki je veliko bolj fleksibilna, kot je npr. pri sistemu *Swarm*. Ker imajo ločen tip za storitev (primer 2-5) od dejanske storitve (primer 2-6), je možno lažje implementirati posamezne implementacije gradnikov v različnih oblakih. V našem primeru bo *Kubernetes* na osnovi anotacij sam vzpostavil razporejevalec *AWB ELB* in uredil, da pravi vsebniki dobivajo zahteve odjemalcev. Takšne podrobnosti so zelo pomembne za produkcijska okolja, saj precej zmanjšajo režijske stroške, potrebne za upravljanje takih sistemov. Ker so etiket specifične za implementacijo, imamo lahko različna fizična okolja, v katerih poganjamo storitev, ki bo upravljala enako funkcijo. Še ena prednost okolja *Kubernetes* je dodelan sistem za izmenjavo skrivnosti in upravljanje s konfiguracijami.

3. Meritve

Kot je predstavljeno v poglavju o metodologiji, smo za merljive kriterije izbrali naslednje:

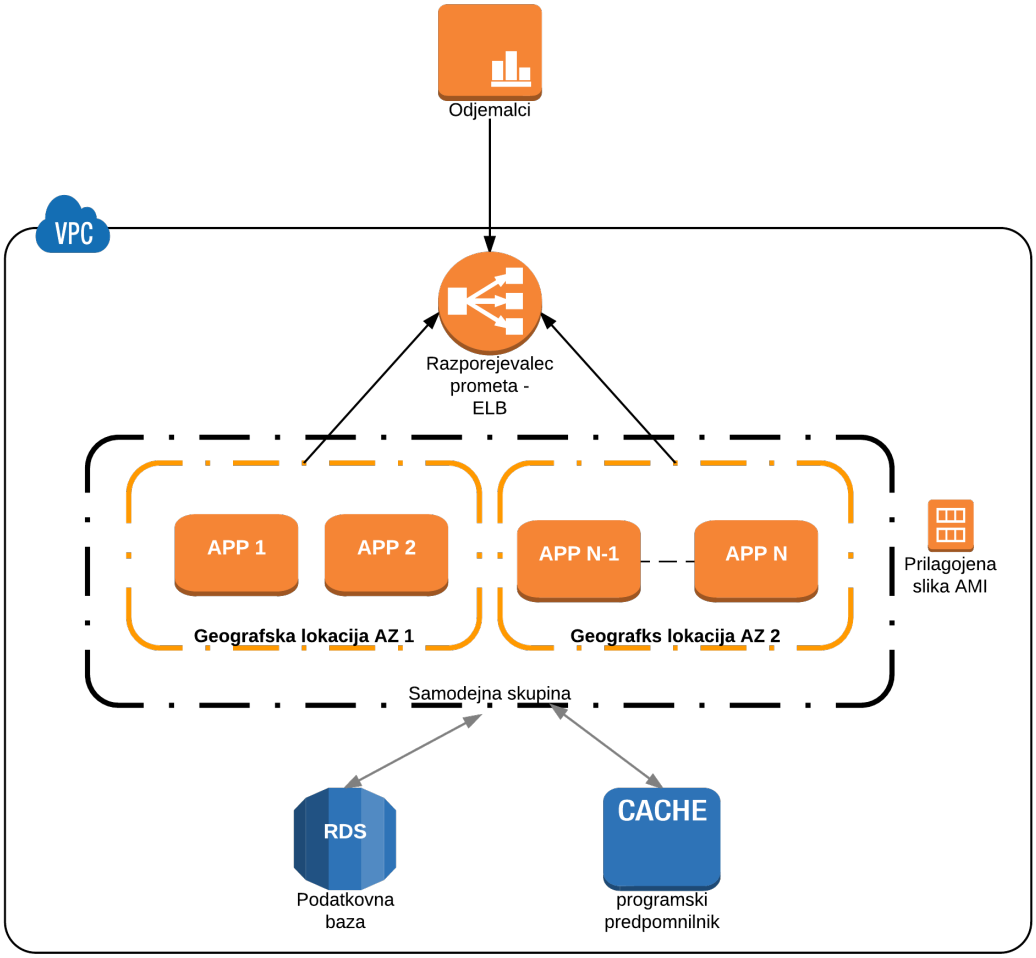
- izkoriščenost virov v odstotkih;
- prepustnost sistema;
- čas, potreben za samodejno odkrivanje storitev;
- čas, potreben za nadgradnjo vsebnikov oz. aplikacije;
- izolacija virov;
- dodajanje novih vozlišč v gručo;
- čas obnovitve storitve po izpadu delovanja.

Z meritvami smo želeli ugotoviti, ali so rezultati meritev primerljivi z rezultati meritev pilotnim okoljem in to, kako so funkcionalnosti, ki so specifične za okolja z vsebniki, primerljive med različnimi orodji. Natančnejši opis in interpretacija meritev sledita v nadaljevanju.

3.1 Opis pilotnega okolja

Za pilotno okolje smo izbrali primer trinivojske spletne storitve, ki je sestavljena iz razporejevalca prometa, aplikacijskega strežnika in podatkovne baze, saj je to največkrat uporabljena struktura v našem podjetju. To je pogosto uporabljena arhitektura oblačne aplikacije [29], ki je zelo značilna za t. i. mikro storitve, razvoj katerih je tudi v našem podjetju čedalje bolj pogost. V našem primeru bomo merili le odzivnost aplikacijskega strežnika, ki je ob povečanem števila uporabnikov “ozko grlo” sistema, saj se tam izvaja večina računskih operacij in je to običajno tudi točka v sodobnih sistemih, v kateri je smiselno horizontalno povečati zmogljivost.

Glede funkcionalnih zahtev na infrastrukturnem nivoju smo za primerjavo vzeli tipično rešitev podjetja, ki jo uporabljamo v oblaku *AWS*. Arhitektura omogoča samodejno odpravljanje napak in samodejno horizontalno prilagajanje računalniških virov glede na količino prometa.



Slika 3-1: Diagram tipične aplikacije *Fairfax* v oblaku *AWS*

Kot je razvidno iz slike 3-1, je aplikacija odjemalcem dostopna posredno prek razporejevalca prometa (ang. *load balancer*) *AWS ELB*, ki razporeja zahteve aplikacijskim strežnikom, ki so del samodejne skupine.

Število strežnikov in zamenjava le-teh ob okvari je samodejna, saj se na podlagi programskih vmesnikov, ki določajo zdravje aplikacije, zasedenosti pomnilnika ali rapoložljivosti procesorske moči gruča povečuje ali zmanjšuje. Prav tako so vsi aplikacijski strežniki znotraj samodejne skupine razdeljeni v dve ali več skupin (odvisno od regije *AWS*), ki so fizično postavljene v različnih strežniških centrih (označeno na sliki z AZ1 in AZ2), da se ob izpadu elektrike, naravnih katastrofah in drugih izrednih dogodkih zagotovi nemoteno delovanje storitev, vendar z zmanjšano zmogljivostjo.

Za upravljanje s slikami aplikacij običajo uporabljamo prilagojene sistemske slike (*Amazon AMI*), ki jih s pomočjo konfiguracijskih sistemov prilagodimo ob prvem zagonu, da imajo pravilno nameščeno različico programa, nastavljene sistemske uporabnike itd., da lahko postanejo del aplikacijske gruče, ki servira podatke končnim uporabnikom. Za naknadne spremembe prav tako skrbi konfiguracijski sistem, ki periodično zagotavlja, da je željeno stanje strežnikov usklajeno.

Naše meritve ne bodo zajemale operacij, kjer je rezultat odvisen od lokalno generiranih podatkov na aplikacijskem strežniku. Večvozliščne gruče vsebnikov trenutno ne omogočajo primerne rešitve prenosa dinamično generiranih podatkov ob izpadu posameznega vozlišča.

Trenutne težave takega okolja so predvsem v neizkoriščenosti sistemskih virov aplikacijskega nivoja, času, potrebnem za zagon novega strežnika ob izpadu ali zahtevi samodejne skupine, ter zagotavljanju enotnega razvojnega in produkcijskega okolja.

Neizkoriščenost sistemskih virov je pogojena z dejstvom, da zagon novega strežnika s pravilno različico storitve lahko traja tudi do dvajset minut. Primaren razlog je, da večina naših aplikacij uporablja interpreterske jezike, ki so zelo odvisni od zunanjih programskih knjižic. To v praksi pomeni, da je ob zagonu novega sistema najprej treba naložiti vse podporne knjižice, prevesti statične odvisnosti, namestiti pravilno inačico programske kode, izvesti integracijske teste in naložiti najbolj pogoste zahteve v pomnilnik. To v praksi pomeni, da zelo težko izvajamo stalno integracijo oz. je ne izvajamo v zadostni meri. Na nekaterih informacijskih sistemih lahko hkrati dela več sto razvijalcev. To pomeni, da bi ob namestitvah, ko imamo več deset strežnikov z dolgoročno tekočo storitvijo in bi radi dnevno neodvisno izdali deset novih funkcionalnosti, za to porabili ves delovni dan.

Za testno aplikacijo smo uporabili storitev, ki preoblikuje lokalne podatke v zapis *JSON* in jih posreduje odjemalcem.

Za programski jezik pilotne aplikacije smo izbrali *Ruby/Rack*, saj gre za priljubljen interpreterski jezik, ki je velikokrat odvisen od zunanjih knjižic, kar pomeni, da je ob novi namestitvi vse odvisnosti treba naložiti s spleta. Tako lahko bolj realno simuliramo potreben čas namestitve aplikacije ob dinamični zahtevi za več novih zmogljivosti.

Vsa testna okolja bodo nameščena v oblaku *AWS*, in sicer v regiji *ap-southeast-2*. Ker v našem primeru ne merimo razpoložljivosti omrežja in funkcionalnosti *CDN*, bodo tudi naši testni odjemalci nameščeni v isti regiji, da težave v omrežju ne bodo vplivale na rezultate.

Ko bomo merili izoliranost virov med storitvami in proces nadgradnje vmesnikov, bomo za referenčno aplikacijo vzeli bolj kompleksen portal, ki je prav tako razvit v okolju *Ruby* in uporablja ogrodje *Rail*.

Za vsa pilotna okolja smo uporabili strežnike v *AWS* z naslednjimi lastnostmi:

- 1 *vCPU*,
- 1 *GB RAM*,
- diskovno polje *SSD*.

Za operacijski sistem vseh pilotnih okolij bomo uporabili *CentoOS 7.2* z jedrom *Linux 3.10.0*.

Za orkestracijska orodja bomo uporabili naslednje inačice:

- *Kubernetes* 1.3.5,
- *Docker Swarm* 1.12.0,
- *Mesos Sphere* 1.7.

3.2 Rezultati testov

V nadaljevanju sledijo opisi uporabljenih metrik, rezultati in interpretacija podatkov.

3.2.1 Izkoriščenost virov

Za meritev izkoriščenosti virov smo uporabili privzeta orodja *Linux*, kot so *psutil*, *ps*, *sar* in *AWS Cloudwatch*. Merili smo zasedenost procesorske moči in pomnilnika. Za ta test nismo uporabili spomina *swap* in nismo merili zasedenosti naprav V/I, saj aplikacija ne shranjuje/dostopa do lokalnih podatkov, ki bi lahko potencialno obremenili kanale V/I.

Ta metrika je zelo pomembna, saj želimo imeti čimvečjo izkoriščenost virov v oblaku, kjer je plačljivi model zastavljen tako, da se plačujejo zakupljeni viri, ne pa dejanska uporaba.

Ko merimo izkoriščenost virov pri uporabi vsebnikov, je treba upoštevati tudi dodatno zasedenost sistemskih virov, potrebnih za delovanje okolja samega, zato je ta meritev opravljena v času mirovanja, ko se na sistemu ne izvaja nobena dodatna operacija ali storitev. V klasičnem okolju teh režijskih stroškov ni, saj poganjamo le eno aplikacijo na posameznem vozlišču. V sistemski uporabi smo upoštevali tudi orodja, ki jih samodejno namestimo kot del standardnega okolja, kot so *New Relic agent*, odjemalec za nadzorovanje delovanja *nagios2* itd.

Okolje	***AWS EC2***	***Kubernetes***	***Swarm***	***Mesos***
Sistemska izkoriščenost (režija)				
***CPU* %**	NaN	2 %	1 %	1–3 %
Pomnilnik %	NaN	450 MB	224 MB	650 MB
Aplikacijska izkoriščenost				
***CPU* %**	10 %	10 %	10 %	10 %
Pomnilnik %	534 MB	534 MB	534 MB	534 MB

Tabela 3-1: Izkoriščenost sistema v primeru mirujočega sistema

Kot je razvidno iz meritev, je dodatna sistemska poraba med orkestracijskimi orodji precej primerljiva in v primeru večjih vozlišč domala zanemarljiva. Vsekakor ima najmanjšo režijsko uporabo klasična aplikacija, saj nima dodatnih spremljevalnih storitev, potrebnih za svoje delovanje. Orodje *Swarm* ima manjšo porabo, saj v primerjavi s *Kubernetes* in *Mesos* ne potrebuje dodatnih spremljevalnih storitev, kot je npr. kolektor dnevniških zapisov. *Mesos* ima pričakovano večjo uporabo, saj uporablja Javin virtualni stroj, ki privzeto zahteva več spomina za delovanje.

Aplikacijska porabe sistemskih virov je med vsemi rešitvami izenačena in se premosorazmerno povečuje s številom odjemalcev. Torej so v primeru aplikacije, kjer imamo konstantno število uporabnikov in primerno velikost vozlišča, razlike med rešitvami zanemarljive.

Prednost vsebnikov je v tem, da lahko v obdobjih nizkega prometa in porabe ene aplikacije sistemske vire koristno uporabimo za sočasno zaganjanje drugih storitev.

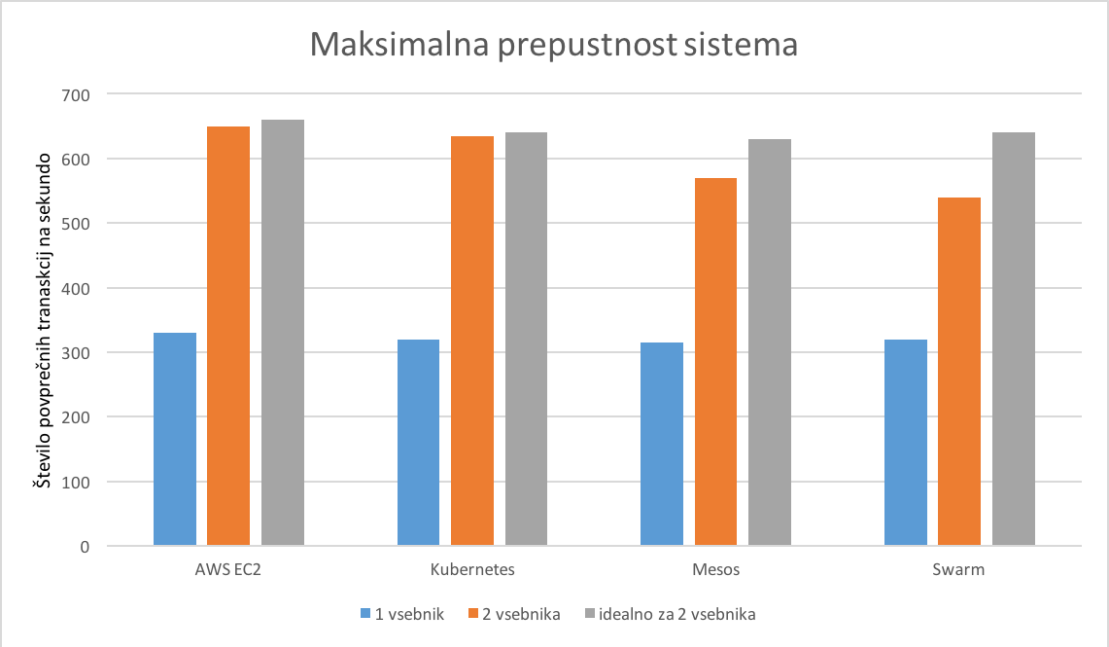
Količina dodatnih vzporednih storitev na vozliščih med orkestracijskimi orodji je enaka, saj vsa uporabljajo *Docker* kot implementacijo vsebnikov, ki uporablja enako tehnologijo za alokacijo spomina in procesorske moči.

3.2.2 Prepustnost sistema

Za ta test smo merili prepustnost sistema na naši testni aplikaciji. Želeli smo izmeriti največje možno število opravljenih transakcij na minuto, ob predpostavki, da je ves promet usmerjen izključno na merjeno storitev. Za meritev smo uporabili orodje *Apache AB*, ki omogoča vzporedno povezavo ter meri povprečen čas za odgovor, število uspešno zaključenih transkacij s pomočjo glave *HTTP*, število nezaključenih transakcij in število transkacij z napako. Za največjo prepustnost bomo vzeli vrednost, pri kateri ne prihaja do napak v sistemu.

Metrika je pomembna, saj lahko z njo ugotovimo ali vsebniki, v primerjavi z virtualnimi stroji, ne glede na dodatno plast sistemske virtualizacije, omogočajo enako prepustnost sistema.

Za ta test bomo merili dva scenarija, in sicer v prvem primeru bomo imeli aplikacijo nameščeno na enem vozlišču, kar pomeni, da bi ob težavah imeli izpad storitve, v drugem primeru pa bomo merili prepustnost, kjer imamo nameščeno aplikacijo na dveh vozliščih. Čeprav s tem testom ne merimo visoke razpoložljivosti storitve, želimo izmeriti, ali večvozliščna namestitev in vmesni razporejevalec prometa vplivata na prepustnost sistema.



Graf 3-1: Največja prepustnost sistema za 1 in 2 vsebnika

Kot je razvidno iz zgornjega grafa, je prepustnost sistema precej podobna naši referenčni aplikaciji. Najboljši rezultat je dosegla storitev v primeru "klasične" namestitve, kar je bilo tudi pričakovati. Preostale storitve v vsebnikih so imele podobne rezultate, pri čemer je *Kubernetes* imel najboljše rezultate, sledila sta mu *Mesos* in *Swarm*.

V namestitvi z dvema vsebnikoma oz. dvema strežnikoma se je po pričakovanjih prepustnost sistema skorajda podvojila. Nepričakovano se je v primeru Swarma/Mesosa povečala tudi odzivnost sistema. Razlog je podobna implementacija razporejevalca, ki je implementirana s podobno tehnologijo, in sicer *nginx* na nivoju vmesnika. V primeru klasične aplikacije, ki uporablja *AWS ELB* in *Kubernetes,* so časi zelo podobni. To lahko pripišemo dejstvu, da za razporejanje prometa *Kubernetes*, nameščen v *AWS*, uporablja enako orodje – *ELB*.

Za slabše rezultate pri Swarmu z dvema vsebnikoma pa lahko najdemo razlog v implementaciji razporejavalca oz. tega, kako odjemalec določi, kateri vsebnik bo stregel vsebino. Proces določanja temelji na zapisih *DNS*, ki odjemalcu naključno vrnejo naslov *IP* vozlišča gruče. To vrednost si za določen čas zapomni tudi odjemalec (ang. *TTL - time to live*), kar v praksi pomeni, da bomo, dokler ne bomo znova zahtevali zapisa s strežnika *DNS*, vedno komunicirali z istim vozliščem. To je lahko pomankljivost, ko imamo večje število strank z enakim posredniških strežnikom (ang. *proxy*), vendar je v praksi to malo verjetno. Do tega problema pri

AWS ELB ne prihaja, saj usmerja promet na osnovi odzivnega časa in zasedenosti strežnikov.

3.2.3 Čas, potreben za samodejno odkrivanje storitev

Za to metriko bomo merili potrebni čas za zaznavanje nove vstopne točke na več vozliščnih namestitvah storitve. To je zelo pomembna in pogosta operacija v okoljih, kjer bi radi prilagodili zmogljivost storitve glede na količino prometa oz. na podlagi zunanjih kazalnikov, kot so zasedenost kopice, ure v dnevu itd. Ta matrika nam pove, kako odzivna so orodja za orkestracijo, hkrati pa lahko razberemo tudi odvečne stroške, ki nastanejo zaradi čakanja na vzpostavitev novega sistema ali dodatno zakupljenih virov.

Za meritev bomo uporabili čas, potreben za zaznavo edinstvene vrednosti v glavi *HTTP*, ki bo identificirala prisotnost novega aplikacijskega strežnika. Za identifikacijo bomo uporabili edinstven identifikator vsebnika. Kljub temu, da bi lahko za meritev uporabili programski vmesnik orodij in upoštevali potrebni čas, da je nova kopija storitve označena kot aktivna, se v praksi dogaja, da to ne odraža dejanskega stanja sistema. Tudi v primeru takih meritev moramo upoštevati, da lahko, zaradi naključnosti razporojevalca prometa, včasih z zamudo ugotovimo prisotnost dodatnega aplikacijskega strežnika.

V primeru referenčne aplikacije bomo povečali zmogljivost za en strežnik, tako da bomo spremenili nastavitve samodejne skupine. Za vsebnike pa bomo povečali število replikacij vsebnikov.

```
# AWS
$ aws autoscaling update-auto-scaling-group --auto-scaling-group-name RUBY_FRI –max-size 2

# Kubernetes
$ kubectl scale –replicas=2 deployments/ruby-example

# Mesos Marathon
$ dcos marathon group scale GROUP_ID 2

#  Docker Swarm
$ docker service scale ruby_fri=2
```

Primer 3-1: Povečanje zmogljivosti vsebnikov na različnih gručah

Kot je razvidno iz rezultatov, je čas, potreben za dodajanje in registriranje novega strežnika v klasični aplikaciji, znatno večji kot pri vsebnikih. Razlog za to je v tem, da slika vozlišča *AMI* nima prednameščenih vseh sistemskih zahtev, tako da mora naknadno pobrati še preostale manjkajoče zahteve in knjižnice, ki so predpogoj za delovanje storitve. Šele nato sistem za upravljanje s konfiguracijami (v našem primeru *Puppet*) označi novonastalo vozlišče kot aktiven element storitve.

	AWS EC2	***Kubernetes***	***Mesos***	***Swarm***
Čas v s	1212	60	60	60

Tabela 3-2: Čas za samodejno odkrivanje nove storitve

Minimalni in maksimalni časi pri vsebnikih so zelo odvisni od trenutne hitrosti registra *Docker*, velikosti slike in podatka, ali je vozlišče že kdaj prej serviralo storitev z enako različico. Če pride do slednjega primera, pomeni, da ima vozlišče že prenameščeno sliko na sistemu, kar pomeni, da ni treba dostopati do registra in prenašati slike. V našem primeru, ko je slika velikosti okoli 800 MB, to lahko traja do petnajst minut. Za našo meritev je bil register postavljen v Ameriki, tako da bi se dalo čase, kljub temu, da so dobri, še izboljšati.

3.2.4 Čas, potreben za nadgradnjo vsebnikov ali aplikacije

V tem testu merimo čas, potreben za nadgradnjo aplikacije, saj imamo dve kopiji vsebnika, zaradi visoke razpoložljivosti, že nameščeni. Prav tako bomo spremljali dosegljivost same aplikacije med tem procesom. To bomo merili s pomočjo orodja *Apache AB*. Podobno kot v prejšnjem primeru bomo za ugotovitev časa namestitve aplikacije v celoti spremljali edinstveno vrednost v glavi zahteve *HTTP* in vrednost programskega vmesnika. V tem primeru bomo za edinstveno vrednost vzeli inačico storitve in bomo merili čas, potreben od začetka procesa do trenutka, ko imajo vsa vozlišča novo vrednost. Metrika je pomembna s stališča stalne integracije in stalne namestitve. Manjši časi so predpogoj za okolje, kjer je mogoče izvajati veliko dnevnih sprememb. To je pomembno za podjetja, kjer bi radi prišli do takega okolja, kjer lahko veliko število razvijalcov dnevno izdaja posodobitve sistemov.

	AWS EC2	*Kubernetes*	*Mesos*	*Swarm*
Čas v s	1200	122	120	70

Tabela 3-3: Čas za nadgradnjo aplikacije za primer dveh kopij storitve

Po pričakovanju so časi skorajda dvakratnik časov, potrebnih za odkrivanje nove storitve. Razlog za to je v tem, da je proces zelo podoben, s to razliko, da je v tem primeru treba na novo namestiti storitev na vsa vozlišča. Pri privzetih nastavitvah vsa orodja nadgradnjo izvedejo zaporedno, kar pomeni, da nadgradnja poteka hkrati samo na enem vsebniku. Ob težavah zagona novega vsebnika se proces nadgradnje prekine. Do malo bolj presenetljivih časov pri Swarmu je prišlo, ker je razporejevalec vsebnikov dejansko dvakrat namestil storitev na enako vozlišče. Takšno delovanje je pričakovano in se lahko zgodi v praksi, kar pa tudi pomeni, da ni treba na novo naložiti slike iz registra, kar znatno pospeši namestitev.

Omembe vredno opažanje v tem primeru je, kako orkestracijsko orodje definira, kdaj je vsebnik zdrav in pripravljen za nudenje storitev zunanjim uporabnikov. To je zelo pomembno poznati, saj morajo nekatere aplikacije ob samem zagonu sprožiti še dodatne korake, kot so inicializacija podatkov, prenalaganje zunanjih vrednosti itd. To v praksi pomeni, da četudi sistemsko storitve/aplikacije delujejo, le-te še niso povsem pripravljene za uporabo.

Žal orodji, kot sta *Mesos* in *Swarm,* nimata sofisticiranega načina za določanje tega, ali je aplikacija dejansko pripravljena servirati podatke. Odločitev, ali je aplikacija pripravljena, določita le na osnovi tega, ali je proces, definiran v sliki, aktiven.

Prednost sistema *Kubernetes* pa je v tem, da je mogoče natančneje določiti stanje storitve. Poleg klasičnega načina lahko določimo tudi dodatno stanje, in sicer pripravljenost (ang. *readiness*). V tem primeru ni dovolj samo to, da je bila slika pravilno nameščena, ampak je tudi potrebno, da uspešno prestane test pripravljenosti, in šele takrat bo dostopna zunanjim uporabnikom.

```
readinessProbe:
  # an http probe
  httpGet:
    path: /health
    port: 8080
    initialDelaySeconds: 15
    timeoutSeconds: 1
```

Primer 3-2: Primer deifnicije pogoja za pripravljenost v *Kubernetes*

V primeru 2-3 bo aplikacija dostopna zunanjim uporabnikom le tedaj, ko bo storitev uspešno vrnila odgovor *HTTP* z vrednostjo 200. Dokler se to ne bo zgodilo, bo *Kubernetes* skril ves promet pred uporabniki. Za naš test smo uporabili privzeti način preverjanja pripravljenosti, ki temelji na prisotnosti procesa v vsebniku.

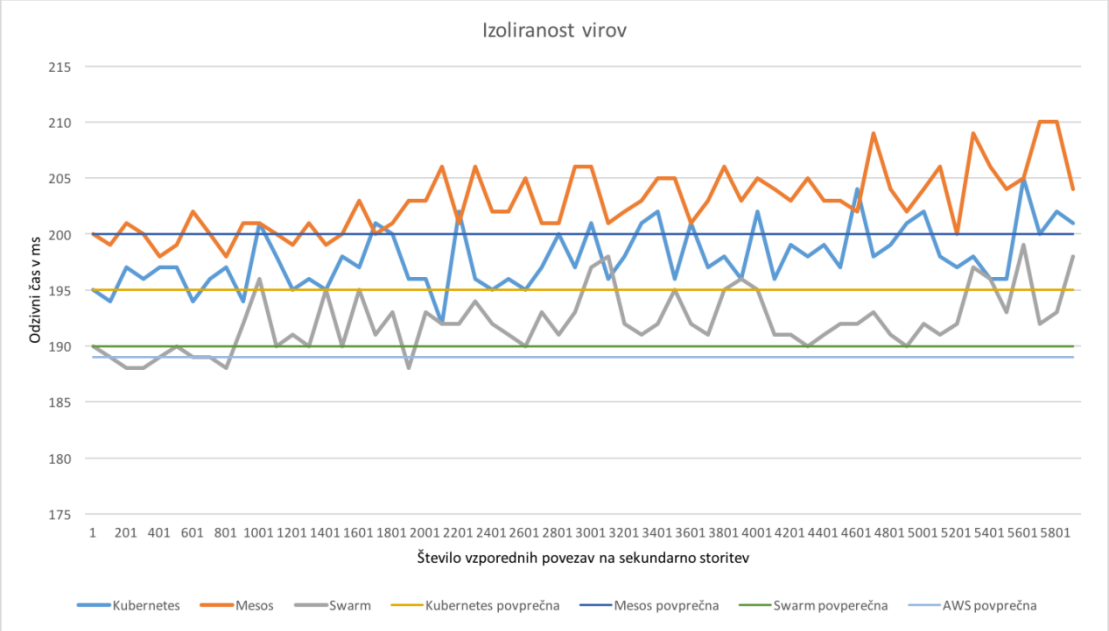
Prav tako *AWS EC2* nima samodejnih načinov izvedbe nadgradnje aplikacij. V praksi sta zelo priljubljena dva načina, in sicer ta, da terminiramo strežnike *EC2* ali uporabimo storitev *AWS CodeDeploy*. V prvem primeru s pomočjo orodij po meri postopoma terminiramo vse strežnike *EC2*, ki so del samodejne skupine, dokler nimamo novih strežnikov, ki s pomočjo sistemov za upravljanje s konfiguracijami namestijo novo izvorno kodo. Prednost takega načina je, da v celoti nadgradimo okolje, vključno z operacijskim sistemom. V drugem primeru, ko uporabimo orodje sistema *CodeDeploy* s skupino *script*, ki sta del izvorne kode, namestimo novo storitev in znova zaženemo storitev z novo kodo. Prednost takega pristopa je, da agenti *AWS CodeDeploy* poskrbijo za koordinacijo med različnimi strežniki. V našem primeru smo merili scenarij, ko terminiramo strežnik.

3.2.5 Izolacija virov

V primeru vsebnikov je zelo pomemben tudi podatek, ali lahko storitve, ki jih izvajamo na skupnih vozliščih, negativno vplivajo na delovanje preostalih storitev. Pri tradicionalnih namestitvah imamo eno storitev na strežnik, tako da se s tem problemom ne srečamo. V primeru vsebnikov pa dejansko želimo imeti več nepovezanih storitev na istem vozlišču, da povečamo izkoriščenost virov. Pri maksimizaciji izkoriščenosti virov je zelo pomembno, da posamična problematična storitev ne onemogoči delovanja preostalih delov sistema. Zelo pomembno je, da

vemo, ali lahko ob nepredvidenih dogodkih pričakujemo slabše delovanje ali celo izpad sistema in s tem povezane stroške.

Za ta test smo merili povprečen odzivni čas storitve, nameščene v vsebnikih, ob sočasnem izvajanju obremenitvenega testa na sekundarni storitvi, ki je nameščena na istem vozlišču. Z obremenitvenim testom smo postopoma povečevali število sočasnih povezav na sekundarno storitev.



Graf 3-2: Odzivni čas glede na obremenjenost gruče za meritev izoliranosti virov

Po pričakovanjih so si rezultati iz grafa 3-2 zelo podobni, saj v vseh primerih uporabljamo enako tehnologijo vsebnikov, ki temelji na funkcionalnosti jedra, in sicer kontrolne skupine (ang. *control group*) in imenskega prostora (ang. *namespace*). Manjša odstopanja nastanejo zaradi samega procesiranja zahtev na nivoju *TCP/IP*, kar je neodvisno od vsebnika in se izvaja na nivoju gostitelja.

3.2.6 Dodajanje novih vozlišč v gručo

V tem delu analiziramo postopek, ki je potreben, da novo vozlišče postane aktiven del gruče, in hkrati merimo čas, ki je za to potreben. Metrika je pomembna, ker se lahko zgodi da v nekaterih trenutkih vse storitve na gruči potrebujejo več računalniških virov, kar pomeni, da moramo povečati število strežnikov. Zaradi poslovnega modela oblačnih storitev ne želimo imeti strežnikov v pripravljenosti, ker bi morali zanje brez

potrebe plačevati. Hkrati pa moramo kljub temu zagotoviti ustrezen nivo kvalitete za primer, ko imamo nepričakovano povečan promet.

Poleg potrebnega časa je tukaj pomemben tudi postopek namestitve programske opreme in avtomatizacija le-tega. To opravilo je potrebno samo za gruče vsebnikov, tako da za ta kriterij ne bomo merili časa referenčne aplikacije.

Ta meritev je pomembna zato, ker je to edini način, kako povečati skupno zmogljivost gruče, in je hkrati del obnovitvnega postopka ob fizičnem izpadu vozlišča.

V našem pilotnem okolju so vsa vozlišča del samodejne skupine *AWS*, tako da v primeru okvare ali zahteve po novem vozlišču sistem samodejno izvede zahtevo za nov strežnik.

```
# Kubernetes
$ kubectl describe nodes

# Mesos Marathon
$ dcos node --json

# Docker Swarm
$ docker node ls
```

Primer 3-3: Ukazi za pridobitev seznama aktivnih vozlišč

Ker je ta operacija skupna, ne glede na orkestracijsko orodje, v tem primeru merimo potreben čas za namestitev in konfiguracijo programske opreme na vozlišču. Podobno kot pri prejšnjih meritvah bomo s pomočjo programskega vmesnika zabeležili, kdaj je gruča zaznala novo vozlišče.

	Kubernetes	***Mesos***	***Swarm***
Čas v s	1.080	1.320	600

Tabela 3-4: Vzpostavitveni časi za novo vozlišče

Kot je razvidno iz rezultatov, je namestitev pri orodju *Swarm* znatno hitrejša. To gre pripisati preprostosti orodja, pri katerem je potrebna le namestitev vmesnika *Docker* in priključitev v gručo. V primeru Kubernetesa in Mesosa pa je poleg Dockerja treba

namestiti vsa spremna orodja in pobrati spremljevalne vsebnike. Ti časi so zelo odvisni od okolja in načina nameščanja programske opreme.

Primerljiva metrika z referenčno aplikacijo je čas za samodejno odkrivanje storitve, torej so ti časi še vedno v sprejemljivih mejah v primerjavi s klasičnimi arhitekturami.

3.2.7 Čas obnovitve storitve po izpadu delovanja

V tem primeru smo merili čas, potreben za obnovitev sistema, in to, kako to vpliva na prepustnost in število napak. Metrika je pomembna s stališča visoke razpoložljivosti, saj bi radi vedeli, koliko časa je potrebno za obnovitev storitve ob različnih izpadih sistema.

Simulirali smo različne scenarije, in sicer naslednje:

- izpad vsebnika,
- prekinjena komunikacija med delovnimi vozlišči in kontrolerjem,
- interno nedelovanje aplikacije.

Za simulacijo izpada vsebnika smo prekinili delovanje vsebnika, ki je bil del skupine storitve. Ob prekinjeni komunikaciji med vozlišči smo na nivoju varnostnih skupin, ki delujejo podobno kot požarni zidovi, blokirali promet *TCP/UDP*. Za zadnji primer pa smo znotraj vsebnika zagnali neskončno zanko, ki je uporabila ves razpoložljiv pomnilnik in proste procesorke cikle. Za to metriko smo merili čas od trenutka, ko orodje zazna napako, do vnovične vzpostavitve polne zmogljivosti sistema.

	Kubernetes	***Mesos***	***Swarm***
Izpad vsebnika	5 s	5 s	5 s
Prekinjena komunikacija	25 s	30 s	10 s
Interno nedelovanje	5 s	7 s	5 s

Tabela 3-4: Čas obnovitve storitve po izpadu

V prvem scenariju gre za enake čase, saj *Docker* samodejno zazna izključeni vmesnik in ga, po privzetih nastavitvah sistema, znova zažene. Časi za interno nedelovanje so podobni, saj *Docker* sproži podobne obnovitvene mehanizme kot pri izpadu vsebnika.

Prav tako ni nobenih presenečenj ob prekinjeni komunikaciji med delovnimi vozlišči in kontrolerjem. Časi so nekoliko višji kot v preostalih dveh primerih, saj mora orodje najprej odkriti, da je vozlišče nedosegljivo, in nato razporediti vsebnike med drugimi aktivnimi vozlišči.

3.3 Rezultati

Glede na zgoraj izvedene meritve so si izbrana orodja precej podobna po rezultatih. Vsa skušajo rešiti podoben problem, ampak na svojevrsten način.

Rezultati niso toliko nepričakovani, saj v ozadju vsi sistemi uporabljajo isto tehnologijo vsebnikov, *Docker*. Razlika pa je v implementaciji distribuiranega sistema in zaznavanju napak. Glede na količino vozlišč, ki smo jih testiral, do bistvenih odstopanj ni prišlo. Za potrebe dejavnosti, ki jih izvajamo znotraj medijske hiše, velikost gruče nikoli ne bo presegla desetih vozlišč, predvsem zaradi zahtev varnostne politike podjetja, ker moramo zaradi različnih *SLA* in *PCI DSS* nekatere storitve imeti v ločenih domenah *AWS*.

Pričakovano so bili rezultati slabši za pilotno aplikacijo v primerih, kjer je bilo potrebno hitro dinamično prilagajanje okolja. To je tehnološka omejitev klasičnih virtualnih strojev in njihovega načina nameščanja in upravljamo z aplikacijami. Vsebniki namreč v celoti ovijejo (enkapsulirajo) aplikacijo, kar pomeni, da je za njeno namestitev treba le pobrati in zagnati pripravljeno sliko.

Bistveno za naše zahteve je, da so vsi orkestracijski sistemi združljivi z našo referenčno aplikacijo in hkrati omogočajo bolj fleksibilno arhitekturo. Zato je bila končna odločitev odvisna predvsem od funkcionalosti orodij.

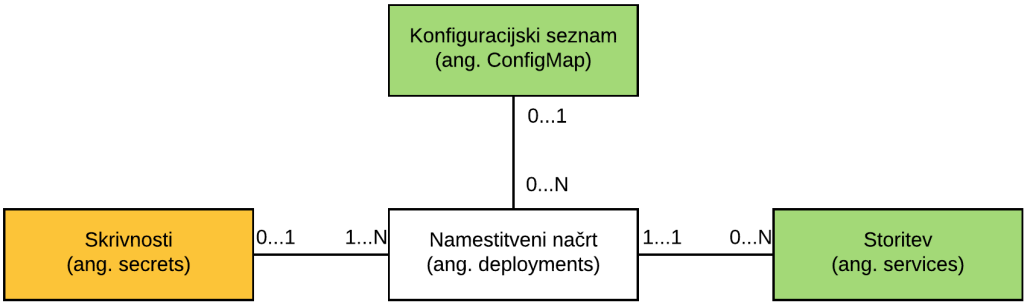
4. Predlog rešitve za medijsko hišo

Po opravljeni analizi rezultatov smo se odločili, da bomo za našo standardno rešitev vsebnikov uporabili orodje *Kubernetes*. V primerjavi s preostalima orodjema je boljši predvsem na področju upravljanja z razporejevalci prometa in ponuja prilagodljive gradnike, ki omogočajo bolj fleksibilno arhitekturo storitev. Elementi, kot sta namestitveni načrt in storitev, omogočajo relativno enostavno integracijo v naše obstoječe okolje AWS ter je hkrati zaradi abstrakcije teh elementov na enostaven način možna migracija v druga oblačna okolja. Prav tako je podpora v skupnosti najboljša v primerjavi s preostalima orodjema. Urejen in dobro dokumentiran proces izdaje nove različice olajša načrtovanje potrebnih inženirskih virov v podjetju.

Alternativna odločitev bi bila vsekakor *Mesos*, še posebej, če bi naše poslovanje temeljilo na serijskih opravilih in analizi velike zbirke podatkov (ang. *big data*) ter bi upravljali z lastnim podatkovnim centrom. Eden izmed največjih pomiselkov tega orodja je kompleksnost namestitve in vzdrževanja le-tega. V trenutni inačici je velik problem, da ni mogoče razporediti skupine vsebnikov na enem vozlišču, kot je to mogoče pri *Kubernetes*. To onemogoča izmenjavo podatkov med vsebniki na nivoju datotečnega sistema in ima lahko performančne posledice v produkciji.

Kljub temu, da so bili rezultati obremenitvenih testov boljši ali vsaj primerljivi s preostalima orodjema, se za *Swarm* nismo odločil, ker manjkajo nekateri ključni elementi, kot sta upravljanje s skrivnostmi in namestitveni načrti. Orodje nima primernega načina za avtentikacijo in je zaradi pomanjkanja domenskega jezika težko slediti spremembam v sistemih za upravljanje s kodo ali z orodji za upravljanje s konfiguracijami. Orodje se zelo hitro spreminja in je pomankljivo podprto s strani oblačnih ponudnikov.

Za standardne elemente naše osnovne arhitekture *Kubernetes* smo uporabili gradnike s slike 4-1.



Slika 4-1: Diagram elementov *Kubernetes*

Vsaka naša storitev je sestavljena iz gradnika skrivnosti, v katerem hranimo vrednosti glede na okolje, kjer želimo storitev poganjati. Za shranjevanje občutljivih podatkov smo razvili manjše neodvisno orodje, ki na osnovi imenskega prostora in veje izvorne kode generira potrebne občutljive podatke. Za takšen korak smo se odločili zaradi varnosti, saj nismo želeli hraniti ključev in gesel v skupnem sistemu za upravljanje izvorne kode. S tem, da smo ločili občutljive podatke od kode in jih izpostavili kot zunanjo storitev, lahko razvijalci neodvisno od operaterjev izvedejo nadgradnjo storitev in vzpostavijo nova razvojna okolja. Primeri skrivnosti so npr. gesla za podatkovno bazo in programski ključi. To je bil tudi predpogoj za uspešno izvajanje stalne integracije.

Konfiguracijski seznam hrani sistemske spremeljivke, kot so ime storitve, frekvenca varnostnih kopij in nastavitve kopice. Vrednosti so drugačne glede na okolje, ampak ker ne vsebujejo občutljivih podatkov, jih lahko hranimo v sistemu za upravljanje s kodo.

Element storitev določa lastnosti razporejevalca prometa, kot so npr. *SSL* certifikat vrata na katerih mora poslušati, itd. Prav tako gre v tem primeru za podatke, ki so odvisni glede na okolje.

Element namestitveni načrt pa določa sliko vsebnika in število kopij vsebnika ter posreduje skrivnosti in konfiguracijske nastavitve neposredno vmesniku.

Vsi ti elementi so abstraktni in jih lahko izmenljivo uporabljamo med različnimi okolji kot so npr. AWS ali Vagrant (orodje, ki omogoča upravljanje z razvojnimi okolji), za razvijalce. Primer konfiguracijskih datotek je dodan v prilogi.

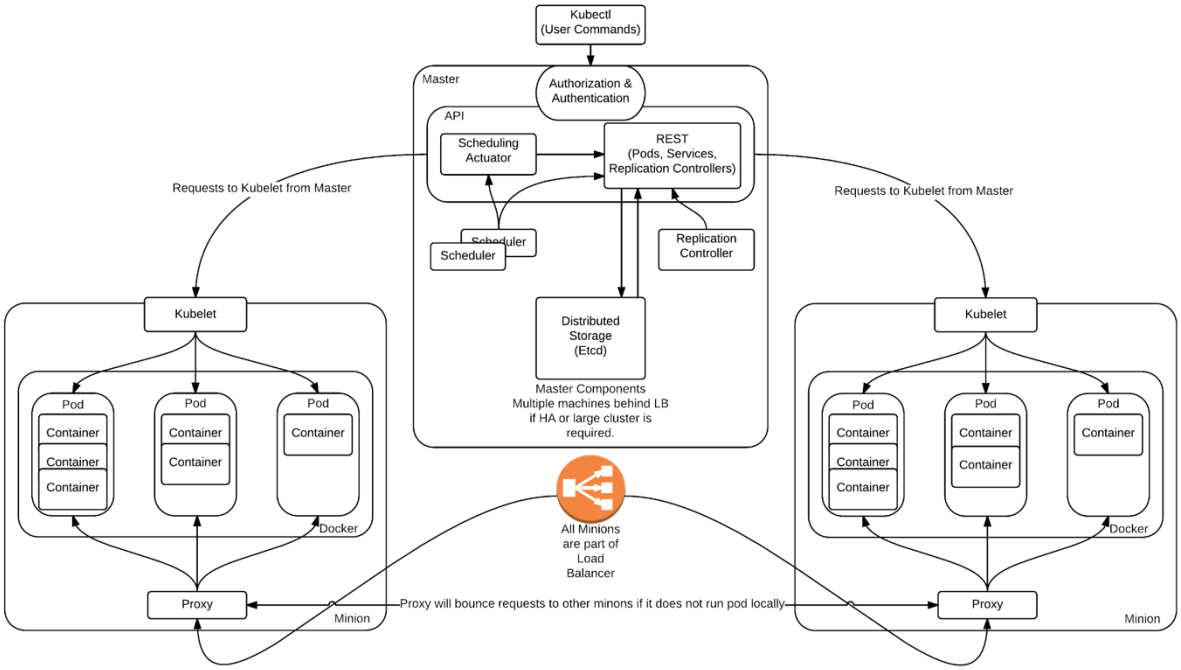
Vsako storitev poganjamo v svojem imenskem prostoru (ang. *Kubernetes namespace*), kar omogoča standardizacijo imen virov, izolacijo konfiguracije in ponovljivost namestitve.

Taka fleksibilna arhitektura omogoča, da lahko neodvisne razvojne skupine pod vodstvom glavnega Scruma (ang. *Scrum master*) določajo svoje neodvisne prioritete glede na zahteve lastnika izdelka (ang. *product owner*). Vsako posamezno zgodbo vodimo kot nalogo znotraj sistema *Atlassian Jira*, ki je neposredno integriran z orodjem *Bitbucket*. Ko je zgodba zaključena, jo lahko razvijalci združijo v svojo razvojno vejo. Vsaka razvojna veja ima vtič, ki samodejno sproži kreacijo nove slike *Docker*. Po uspešni gradnji slik, orodje *Jenkins* samodejno namesti novo sliko v gručo *Kubernetes* v skupinskem imenskem prostoru. To pomeni, da je namestitev v predrazvojno okolje samodejna, ponovljiva in neodvisna od sistemskih inženirjev.

Ko želimo izdati novo funkcionalnost v produkcijsko okolje, je potrebno spremembo samo združiti na glavno (ang. *master*) vejo sistema za upravljanje s kodo, orodji *Jenkins* ali *Codeship* nato samodejno in postopno namestita novo sliko na vsa vozlišča v produkciji. Sledljivost spremembam in komunikacija z lastniki produktov je transparentna, saj je na podlagi imena veje, katerega združimo z glavno vejo, mogoče identificirati zgodbo in preveriti njen opis.

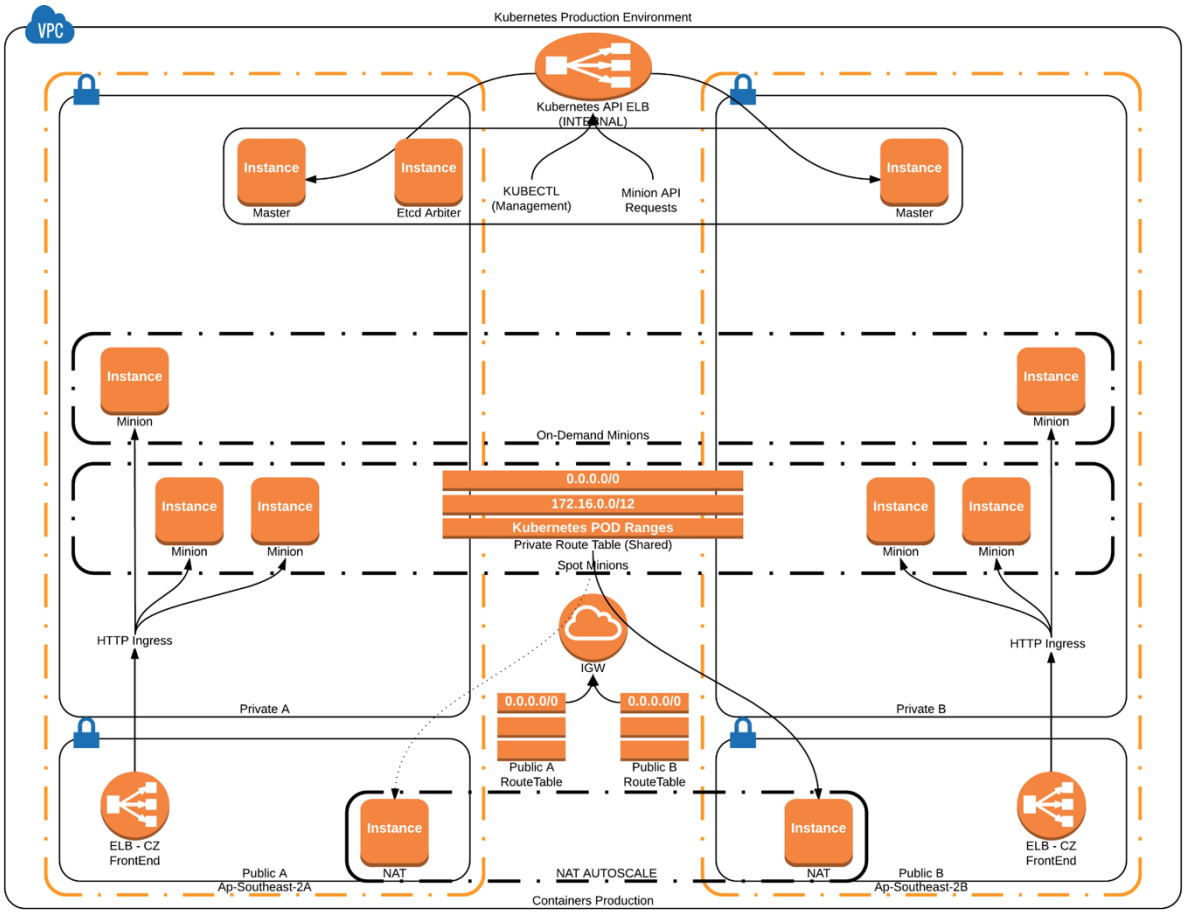
Pri načrtovanju infrastrukture smo uporabili deklarativni jezik *AWS Cloudformation*, ki omogoča definicijo infrastrukture kot kode. S tem smo želeli imeti definirano celotno okolje s pomočjo kode, kar omogoča visoko stopnjo ponovljivosti in posledično hitro nadgradnjo okolij. Ta zahteva je bila za nas zelo pomembna, saj je *Kubernetes* kot preostala orkestracijska orodja za vsebnike še v fazi hitrega razvoja in se zelo hitro spreminja. V praksi to pomeni, da bomo morali v naslednjih mesecih pogosto migrirati storitve med gručami, da bi lahko v celoti izkoristili vse napredne funkcionalnosti, ki jih prinašajo nadgradnje.

Za poenostavitev razvoja in izboljšanje kvalitete naših storitev smo vzpostavili dve identični okolji, ki se razlikujeta le po zmogljivosti računalniških virov in omrežju, zaradi različnih varnostnih zahtev. To je tudi ena izmed zahtev 12-faktorske metodologije, in sicer ta, da so si okolja čim bolj podobna, kar v razvojnem procesu omogoča zgodnje identificiranje potencialne težav.



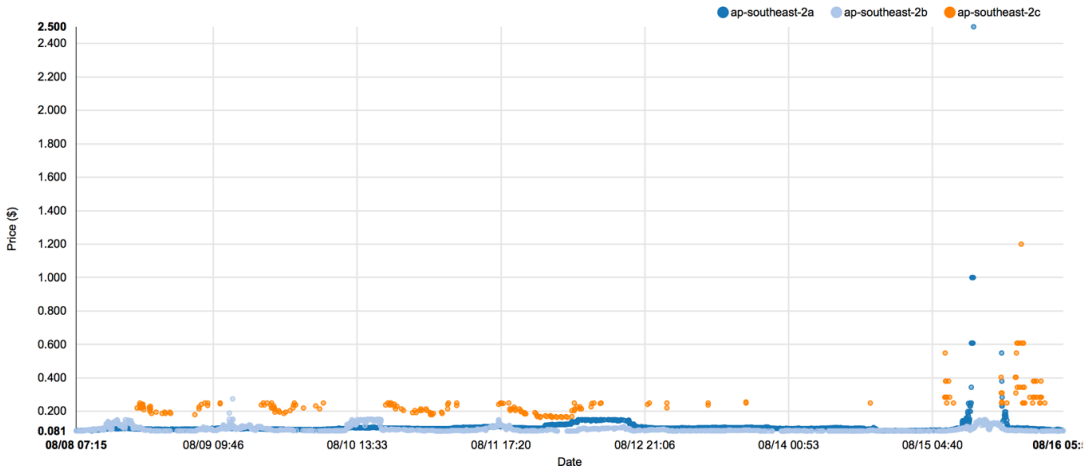
Slika 4-2: Arhitektura namestitve *Fairfax Media Kubernetes* v oblaku *AWS*

Kot je razvidno iz slike, smo delovna vozlišča namestili kot del samodejne skupine, da izkoristimo prednosti, ki jih ta prinaša. Zaradi navedenega lahko ob izpadu posameznih vozlišč in potrebe po novih virih samodejno dodamo nove zmogljivosti.



Slika 4-3: Natanča namestitev okolja *Kubernetes* v *AWS*

Zaradi zmožnosti hitrega dodajanja novih vozlišč smo zastavili tudi tako arhitekturo, da imamo dve podobni samodejni skupini, ki zagotavljata delovna vozlišča. Bistvena razlika je v tem, da ena skupina uporablja strežnike na zahtevo (ang. *spot instances*). To pomeni, da se cena prilagodi glede na povpraševanje in ponudbo.



Graf 4-1: Cena strežnika *c4.2xlarge* v regiji *AWS ap-southeast-2*

Kot je razvidno iz grafa, je tak tip virtualnega stroja bistveno cenejši, saj polna cena znaša 0,54 $/h, kar pomeni, da v povprečju prihranimo 50 % ali več glede na zakupljene vire. Edina težava pri taki zasnovi je, da lahko ob nadpovprečni zahtevi po določenem tipu virtualnega stroja izgubimo večje število vozlišč. Na osnovi empiričnih podatkov smo prišli do zaključka, da 60 % naših vozlišč uporablja standardne virtualne stroje in preostalo na zahtevo.

Kljub temu, da *Kubernetes* z vsebniki omogoča elegantno rešitev za večino naših problemov, še vedno obstajajo izjeme. Storitev, kot so *ElasticSearch*, *Cassandra* oz. dolgoročne tekoče storitve, ki so običajno veliki porabniki pomnilnika in procesorske moči, ni smiselno seliti v vsebnike. Režijski stroški, povezani z upravljanjem gruče, in konstatna uporaba virov bi prinesli samo dodatne nepotrebne stroške in morebitne težave ob izpadu sistema. Take storitve se redkeje nadgrajujejo, tako da so prednosti, ki jih prinaša *Kubernetes* pri stalni integraciji, zanemarljive. Trenutna rešitev, ki temelji na *AWS ASG*, *AWS Cloudformation* in *AWS Cloudwatch*, je dovolj dobra in ekonomsko ugodna, da migracija za produkcijo ni smiselna. Seveda obstajajo izjeme, kot so npr. nekatera preprodukcijska okolja, kjer uporabljamo *ElasticSearch*. Zaradi ekonomskih razlogov in števila potrebnih podobnih okolij imamo manjše namestitve v Kubernetesu.

Tudi storitve, ki temeljijo na javanskem programskem jeziku in imajo napovedljiv promet ter se redko nadgradujejo, ni nujno da so primeren kandidat za premestitev v vsebnike, predvsem zaradi dejstva, da je omejevanje dodeljenega spomina in prioritete procesorske moči za javanski virtualni stroj enostavno opravilo. To pomeni, da lažje dosežemo dober izkoristek virtualnega stroja, kar je za podjetje najbolj ekonomično.

4.1 12-faktorska aplikacijska zasnova

V nadaljevanju bomo strnjeno opisali, kako uporaba vsebnikov in orodja *Kubernetes* omogoča okolje za uporabo 12-faktorske metodologije razvoja aplikacij. Te smernice so nastale kot posledica izkušenj različnih podjetij z upravljanjem tekočih spletnih storitev v različnih okoljih.

4.1.1 Koda

Koda je hranjena v enotnem sistemu za upravljanje izvorne kode in vsaka slika *Docker* je samodejno ustvarjena iz nje. Za označevanje inačice uporabljamo zgoščeno vrednost *SHA1*, ki je edinstvena vrednost na osnovi vsebine datotek in strukture map v repozitoriju *Git*. Na takšen način lahko vedno nedvoumno sledimo spremembam aplikacije in imamo ponovljive korake, da lahko pridemo do enakih rezultatov.

4.1.2 Odvisnosti

Že po zasnovi *Docker* omogoča ovijanje (enkapsulacijo) aplikacije znotraj vsebnikov, kar omogoča neodvisno delovanje aplikacije od operacijskega sistema oz. gostitelja. To pomeni, da bo naša aplikacija lahko delovala v vseh okoljih, kjer lahko poganjamo *Docker*. To dosežemo tako, da vse potrebne odvisnosti namestimo med procesom kreiranja slike kot zaporedje ukazov v *Dockerfile*.

V praksi je tako, da slike naših sistemov gradimo na podlagi prosto dostopnih osnovnih slik, kot so *Ubuntu16*, *CentOS7* itd. Te slike so generične in podpirajo širok nabor sistemskih knjižic. Posledica tega je, da so privzeto lahko slike veliko večje, kot je dejansko potrebno. Nekatere naše končne slike *Ruby/Rails* so lahko npr. velike tudi do 3 GB, vključno z vmesnimi slikami. Kljub temu, da si veliko slik deli skupne plasti zaradi lastnosti datotečnega sistema *AUFS*, to lahko negativno vpliva na čas, potreben za prvi zagon vsebnika.

4.1.3 Konfiguracijske datoteke

Konfiguracijske datoteke ne smejo nikoli biti shranjene v sistemu za upravljanje izvorne kode in morajo biti edine spremenljivke, ki so drugačne glede na okolje, kjer bi radi poganjali aplikacijo. To je predpogoj za vzpostavitev podobnih produkcijskih in predprodukcijskih okolij, saj imamo lahko enotne aplikacijske slike, ki jih prilagodimo s pomočjo sistemskih spremenljivk. To tudi poenostavi testiranje aplikacij, saj lahko enako sliko uporabljamo v testnih kot tudi v produkcijskih okoljih.

Privzeto *Docker* že omogoča izpostavitev konfiguracijske spremenljivke s pomočjo sistemskih spremenljivk, vendar bi bilo to v večjih sistemih kmalu neobvladljivo. Zato v našem primeru uporabljamo konfiguracijske sezname in skrivnosti *Kubernetes*, ki na bolj sistematičen način rešijo to težavo.

```
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: elasticsearch-config
data:
  cluster.name: example
  service: “elasticsearch-transport”
```

Primer 1-1: Primer konfiguracijskega seznama *Kubernetes*

4.1.4 Podporne storitve

Podporne storitve so storitve, ki jih aplikacija uporablja za delovanje. Primer takih storitev so podatkovni strežniki (*MySQL*, *PostgreSQL* itd.) ali aplikacijski predpomnilniki (*redis*, *memcache* itd.).

Zahteva za 12-faktorsko metodologijo razvoja je, da aplikacija obravnava podporne storitve kot dodatne vire, torej mora omogočati, da aplikacija lahko brez spremembe kode doda ali odstrani storitev. Prav tako ne sme biti razlike med lokalno ali zunanjo storitvijo. To omogoči lažjo prenosljivost in izboljša homogenost aplikacije.

V tem primeru *Docker* oz. *Kubernetes* neposredno ne pomagata pri izboljšanju neodvisnosti podpornih storitev, saj je ta večinoma odvisna od tega, kako implementiramo aplikacijo. Kljub temu, da imajo slike *Docker* omejeno količino datotečnega prostora in se vsebniki lahko pogosto menjujejo oz. je razporejanje med vozlišči pogosta operacija, prisili razvijalce k zasnovi takega sistema, da lahko brez sprememb kode izmenjujemo podporne storitve.

4.1.5 Zgradi, objavi, poženi

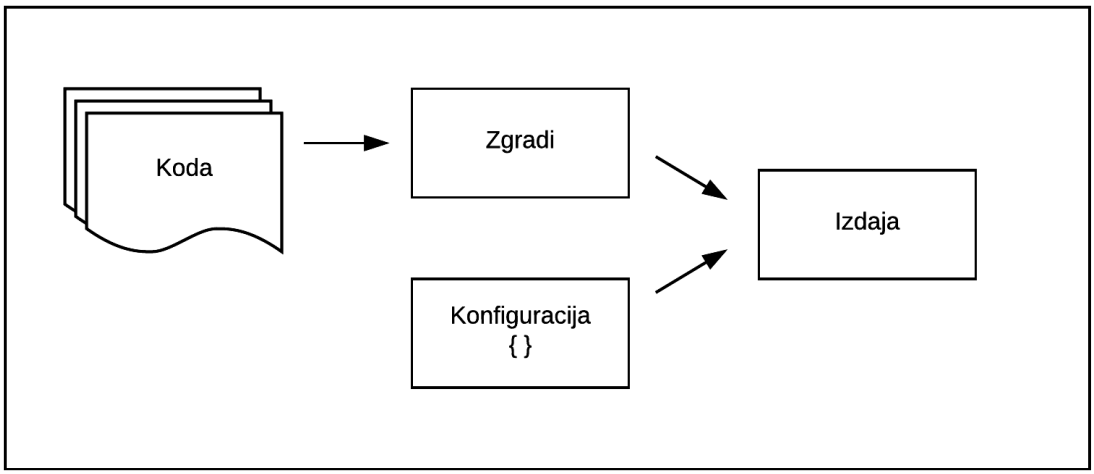
12-faktorska aplikacija strogo loči process prevajanja, pakiranja in poganjanja aplikacije.

V fazi prevajanja mora sistem s pomočjo sistema za upravljanje izvorne kode pobrati izvirno kodo, pridobiti in prevesti odvisnosti ter primerno označiti inačico končnega rezultata operacij – največkrat binarna koda ali končna sredstva (ang. *assets*).

V fazi pakiranja sistem združi binarno kodo ali sredstva, pridobljena iz faze prevajanja s konfiguracijskimi vrednostmi, in pripravi okolje do te mere, da je mogoče poganjati aplikacijo.

V fazi poganjanja (ang. *runtime*) sistem zažene zapakirano aplikacijo v danem okolju, tako da poskrbi, da je proces aplikacije aktiven.

Takšna ločitev procesa omogoča lažji nadzor življenjskega cikla aplikacije in omogoča ponavljajoče se izboljšave posameznih faz.



Slika 4-4: Elementi različnih faz življenjskega cikla aplikacije

V našem primeru smo ločili postopke tako, da smo za fazo prevajanja uporabili kombinacijo orodja *Jenkins* in sledenja sprememb na izvorni kodi, ki kot rezultat kreirajo sliko sistema *Docker*, ki jo shranimo v register *Docker*.

Za fazo pakiranja in poganjanja pa skrbi *Kubernetes* z gradniki, kot so storitev, namestitveni načrt in etikete.

4.1.6 Procesi

V tem primeru se priporoča, da so vsi procesi aplikacije neodvisni in da ne hranijo nobenih lokalnih podatkov oz. da niso odvisni od seje med odjemalcem in strežnikom. Vse trajne podatke moramo hraniti v podatkovni bazi oz. v podpornih storitvah (gl. 4.1.4). Predpostavka je, da lahko podatki, ki so v pomnilniku ali na lokalnem disku, kadar koli izginejo. Idealno se vsi potrebni podatki za delovanje sistema izmenjajo v eni seji med uporabnikom in aplikacijo.

Taka zahteva omogoča visoko razpoložljivo arhitekturo storitve, saj lahko poljubno dodajamo oz. odstranjujemo zmogljivosti aplikativnega nivoja. Pri tej zahtevi tudi *Docker* ne more neposredno pomagati, saj je vse odvisno od razvijalca aplikacije, vendar jih v tako arhitekturo prisili zaradi izmenljive narave vsebnikov.

4.1.7 Povezava vrat

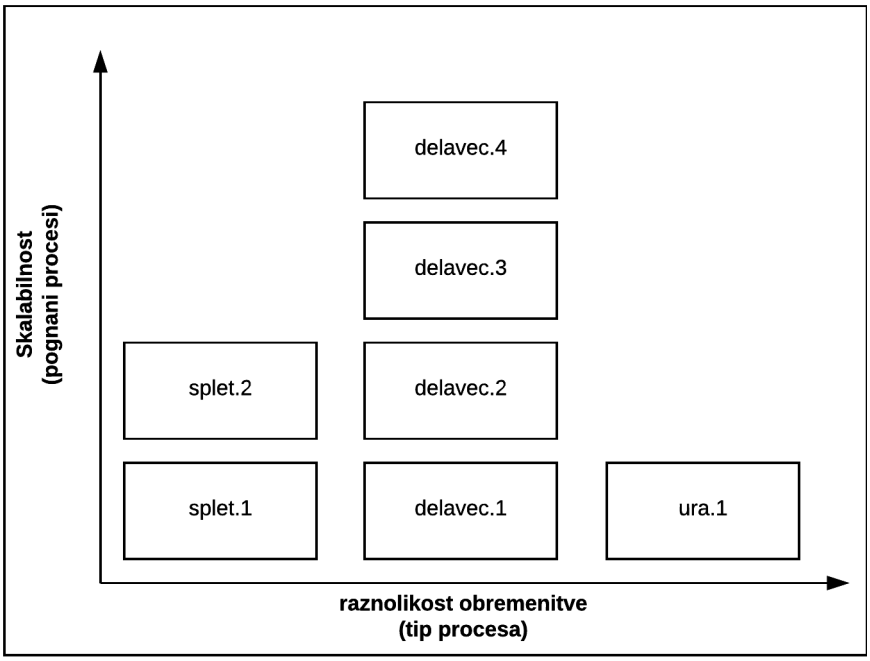
Vse aplikacije, ki sledijo 12-faktorski metodologiji, morajo biti neodvisne, kar pomeni, da ne smejo biti neposredno odvisne od preostalih storitev. Primer take uporabe je npr. odvisnost aplikacij *php/ruby* od spletnega strežnika *Apache*. Treba je zagotoviti, da poteka komunkacija z aplikacijo prek vrat, neposredno, brez vmesnika.

To pri razvoju sodobnih aplikacij ni več težava, saj ima večina programskih jezikov ogrodja, ki neposredno implementirajo protokol *HTTP*.

Privzeto tudi *Kubernetes* in preostala orodja omogočajo neposredno izpostavitev vrat na vsebniku. *Kubernetes* olajša to opravilo, tako da s pomočjo elementa storitev samodejno preusmerja promet na pravi vsebnik.

4.1.8 Sočasnost

Aplikacijo je treba načrtovati tako, da je mogoče povečati sočasnost s pomočjo horizontalnega povečanja zmogljivosti. Ta zahteva je neposredno povezana z zahtevo o procesih (4.1.6) in zahtevo o neodvisnostih okolja (4.1.2). To dosežemo tako, da uporabimo procesni model. Z uporabo tega modela razvijalci lahko zasnujejo sistem tako, da razporedijo promet glede na vrsto opravila med specializiranimi procesi.



Slika 4-5: Procesni model povečanja prepustnosti

Takšen model ne sme nikoli poslati procesa v ozadje (ang. *daemonize process*), ampak mora prepustiti upravljanje operacijskemu sistemu.

Prav zaradi takih zahtev smo se odločili uporabiti orkestracijsko orodje *Kubernetes*, ki že v osnovi podpira horizontalno povečanje zmogljivosti s pomočjo namestitvenih načrtov in replikacijskih kontrolerjev.

4.1.9 Enkratna uporaba procesa

Procese lahko enkratno uporabimo. To pomeni, da jih lahko poženemo ali zaustavimo po želji, kar omogoča hitrejšo izdajo, poenostavljeno horizontalno povečanje zmogljivosti, itd. Vsak proces mora omogočati hitro in pravilno zaustavitev procesa, če pošljemo zaključni (ang. *sigterm*) signal. Posledica tega je, da so aplikacije bolj robustne.

Kubernetes ali *Docker* neposredno ne rešita tega problema, ker je to odvisno od tega kako implementiramo aplikacijo. Kljub temu pa smernice razvoja slik vsebnikov priporočajo uporabo enega procesa na vsebnik,

4.1.10 Enakost predprodukcijska/produkcijska okolja

Enaka binarna koda ali končna sredstva morajo biti uporabljena v predprodukcijskih in produkcijskih okoljih. To omogoča poenostavljen razvoj, izboljša postopke testiranja in poveča zaupanje v sistem.

V tem primeru so vsebniki pomembni ne glede na to, katero orkestracijsko orodje uporabimo, saj so slike aplikacij neodvisne in enake ne glede na okolje.

4.1.11 Dnevniški zapisi

Dnevniški zapisi morajo biti obravnavani kot vir dogodkov. Aplikaciji ni treba skrbeti za dnevniške zapise in njihovo upravljanje (rotacija, arhiviranje). Vse dogodke lahko shranimo na standardni izhod in zunanji sistemi bodo poskrbeli za analizo. S takšnim načinom lahko zunanji sistemi glede na potrebo skrbijo za pravilno upravljanje z dnevniškimi zapisi. V produkcijskih okoljih so zahteve precej drugačne kot v razvojnem okolju, kar pa mora biti skrito pred razvijalci.

Docker in *Kubernetes* privzeto omogočata, da so vse vrednosti, zapisane na standardni izhod, vidne prek konzole, in da je mogoče prek programskih vmesnikov preusmeriti zapise v zunanje sisteme. V našem primeru smo se odločili za uporabo rešitve *ELK*, ki s pomočjo elementa *Kubernetes daemon set* zagotovi, da vse vrednosti, zapisane na *STDOUT*, shranimo na gručo *ElasticSearch*, kjer jih je možno analizirati s pomočjo orodja *kibana*.

4.1.12 Proces upravljanja

Tradiocionalno so operacije upravljanja ločene od dolgo pognanih procesov. Velikokrat bodo razvijalci razvili podporna orodja, ki izvajajo enkratna upravljalna vzdrževalna opravila. Zaheva 12 faktorske metadologije je, da enkratna ali podobna nestandardna opravila morajo biti izvedena v enakem okolju in z enakimi konfiguracijskimi nastavitvami kot dolgoročni procesi. Dodatna upravljalna opravila morajo biti del enake izvorne kode. Enaka izolacijska tehnika mora biti uporabljena za vse procese. Npr. v primeru jezika *Python* morajo biti vse operacije in procesi pognani v enakem virtualnem okolju, da zagotovimo enake sistemske in aplikacijske zahteve.

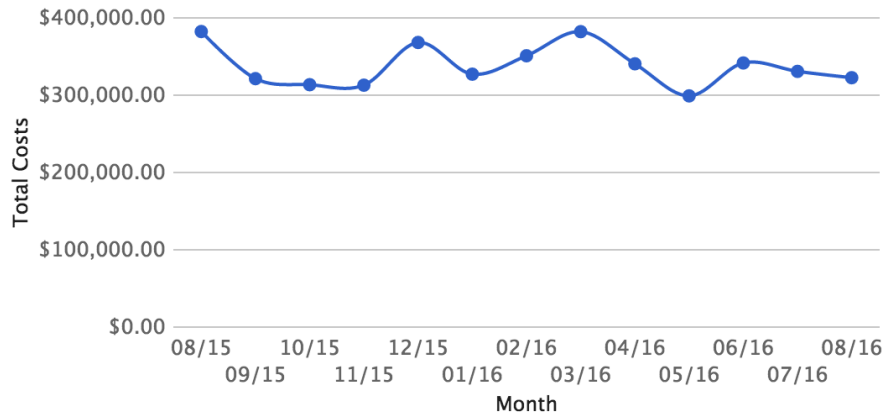
Vsebnik privzeto izolira (enkapsulira) vse procese v enako okolje. To omogoča enako razvojno/produkcijsko okolje, ki zmanjšuje možnosti napak.

5. Sklepne ugotovitve

Kot je razvidno iz testov delovanja, ni velikih razlik med izbranimi orkestracijskimi orodji, saj vsa v ozadju uporabljajo enako tehnologijo vsebnikov *Docker*. Prav tako so si rezultati med vsebniki v primerjavi s tradicionalnimi *AWS EC2* virtualnimi stroji po merljivih zmogljivostih zelo podobni. Bistvene razlike so v funkcionalnosti orodij in prednostih, ki jih prinašajo podjetjem, kjer so stalna integracija, zmanjševanje razlik med okolji in poenostavljeno upravljanje pomembni. Vsebniki omogočajo izboljšano distribucijo ter hitrejšo in lažjo izdajo novih inačic programske opreme.

Razvijalci so se v podjetju hitro prilagodili novemu sistemu, predvsem pa jim je bila všeč podobnost med razvojnim in produkcijskim okoljem. Hkrati smo uspeli razbremeniti delo sistemskih inženirjev, saj je bilo tradicionalno upravljanje z operacijskim sistemom, namestivijo sistemskih knjižic itd. zaradi varnosti zunaj domene razvijalcev.

Iz ekonomskega stališča je takšna rešitev tudi ugodna, ker smo uspeli stabilizirati stroške uporabe, kljub temu da smo povečevali število storitev v naših sistemih.



Graf 5-1: Stroški računa *AWS* v podjetju

Vsekakor je področje persistentnih podatkov tisto, na katerem bo potrebno še nekaj dela, saj trenutne reštitve ne omogočajo visoke razpoložljivosti in niso dovolj odzivne pod visoko

obremenitvijo. Trenutne rešitve, kot so *Ceph*, *Flocker* itd., niso še primerne za produkcijska okolja, saj so zaradi slabo definiranih programskih vmesnikov še zelo nestabilne.

Bolj natančno smo testirali vtič, ki uporablja *AWS EBS* kot medij za persistentne podatke. Deluje tako, da ob terminaciji poda oz. zahteve za premestitev na drugo delovno vozlišče kontrolno vozlišče s klicem do programskega vmesnika *AWS* premesti sliko *EBS* na drugi strežnik. Težava pri tej implementaciji je, da je proces premestitve počasen zaradi tega, ker moramo počakati, da operacijski sistem pravilno zaključi vse odprte seje do podatkovnega sistema. Druga težava pa je, da je zaradi omejitve implementacije *AWS* diskovnega polja *EBS* mogoče premeščati slike samo znotraj enega dostopnega območja (ang. *availability zone*). To pomeni, da bi morali imeti vsa naša delovna vozlišča znotraj enega dostopnega območja, kar bi onemogočilo visoko dosegljivost ob izpadu posameznega podatkovnega centra znotraj regije. Alternativen pristop bi bil kreiranje nove slike *EBS* na osnovi posnetka diskovnega polja, vendar bi bilo takšno opravilo predolgo, saj je premosorazmerno z velikostjo podatkov.

Vsekakor je ena pomanjkljivost vsebnikov tudi ta, da podpirajo le operacijski sistem *Linux*. V našem primeru to ni bila omejitev, vendar bi v okoljih, kjer prevladuje okolje *Windows*, vsebniki vsekakor odpadli.

Še eno področje, ki bi se ga dalo izboljšati, je samodejna migracija vsebnikov ob dodajanju vozlišč. Pri vseh orodjih bo razporejevalec le na začetku zagona nove naloge oz. ob izpadu vozlišča znova razporedil vsebnik med razpoložljivimi vozlišči. V praksi to pomeni, da bodo nove zmogljivosti, ki jih dodamo gruči, ostale neuporabljene, dokler ne zahtevamo novih nalog.

Zanimivo je tudi spremljati nekatere preostale rešitve, kot so denimo *RedHat OpenShift*, ki za osnovo uporablja *Kubernetes*, ampak z dodatnimi funkcionalnimi nabori. Predvsem je zanimiva njihova dodelana rešitev na področju avtentikacije in komercialna podpora, kar je zelo pomembno za večja podjetja.

Razvojna okolja in skupine razvijalcev so v zadnjih letih postale zelo heterogene. Znotraj večjih podjetij, primer takega je tudi naša medijska hiša, imamo lahko tudi deset neodvisnih razvijalskih skupin, ki imajo visok nivo avtonomije pri izbiri programskih jezikov in orodij za učinkovito doseganje ciljev. Z uvedbo agilnih metodologij razvoja se je cikel dodajanja funkcionalnosti in izdajanja inačic programske opreme bistevno skrajšal, veliko je skupin, pri katerih cikel traja vsega dva tedna. Iz velikih, homogenih in kompleksnih sistemov se je zgodil prehod na mikrostoritve. Senzacionalna naravnanost medijskega sveta in viharno okolje spletnih

storitev velikokrat povzroči nepričakovani porast zahtev za storitev, ki je do nedavnega mirovala. Jasno je postalo, da klasična arhitektura togih fizičnih sistemov in tudi virtualnih strojev takim nalogam nista kos.

Tehonologija vsebnikov se v takih okoljih pokaže kot zelo obetavna, saj ponuja rešitev, ki bistevno razbremeni upravljalce sistemov ter omogoča poenoteno razvojno in produkcijsko okolje. Samodejna replikacija ob izpadih ter dodajanju in odvzemanju vozlišč v pogojih spremenljivega prometa omogočata optimalno in ekonomično izkoriščenost strojnih virov ter posledično ekonomično poslovanje podjetja. V našem primeru se je pokazalo, da je za nekatere storitve ta prehod smotrn in upravičen.

6. Viri

[1] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, Docker Containers across Multiple Clouds and Data Centers, 2015. IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), IEEE, 2015, str. 368–371.

[2] J. Turnbull, The Docker Book: Containerization is the new virtualization, James Turnbull, 2015.

[3] K. Matthias, S. P. Kane, Docker Up and Running: Shiping reliable containers in production, O'Reilly, 2015.

[4] D. Bernstein, Containers and cloud: From LXC to Docker to Kubernetes, IEEE Cloud computing 2325-6095/14, 2014, str. 81–84.

[5] S. Newman, Building microservices: Designing fine-grained systems, O'Reilly, 2015.

[6] 12 Fakotorska metodologija razvoja aplikacij. Dostopno na https://12factor.net

[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, Google Inc., 2015.

[8] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, M. Steinder,
Performance Evaluation of Microservices Architectures Using Containers, Network Computing and Applications (NCA), 2015. IEEE 14th International Symposium, 2015, str. 27–34.

[9] J. Walter, V. Chaudhary, M. Cha, S. Guercio, S. Gallo, A comparison of virtualization technologies for hpc, v: Advanced Information Networking and Applications, AINA, 2008. 22nd International Conference, 2008, str. 861–868.

[10] T. Adufu, J. Choi, Y. Kim, Is container-based technology a winner for high perfor- mance scientific applications?, v: Network Operations and Management Symposium (APNOMS), 2015. 17th Asia-Pacific, 2015, str. 507–510.

[11] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based ope- rating system virtualization: A scalable, high-performance alternative to hypervisors, SIGOPS Oper. Syst. Rev. 41 (3) (2007), str. 275–287.

[12] M. Gomes Xavier, M. Veiga Neves, C. Fonticielha de Rose, A performance compa- rison of container-based virtualization systems for mapreduce clusters, v: Parallel, Distributed and Network-Based Processing (PDP), 2014. 22nd Euromicro Internati- onal Conference, 2014, str. 299–306.

[13] R. Dua, A. Raja, D. Kakadia, Virtualization vs containerization to support paas, v: Cloud Engineering (IC2E), 2014. IEEE International Conference, 2014, str. 610–614.

[14] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, IBM Research Division, Tech. Rep. RC25482, 2014.

[15] KVM hypervisor. Dostopno na http://www.linux-kvm.org/page/Main_Page

[16] Docker company history. Dostopno na https://www.docker.com/company

[17] M. Kerrisk, Control group configuration unit settings: http://man7.org/linux/man-pages/man5/systemd.cgroup.5.html [Pridobljeno: 13. 2. 2016.]

[18] M. Kerrisk, Control group configuration unit settings: http://man7.org/linux/man-pages/man7/systemd.namespaces.7.html [Pridobljeno: 13. 2. 2016.]

[19] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, C. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, v: Parallel, Distributed and Network-Based Processing (PDP), 2013. 21st Euromicro International Conference, 2013, str. 233–240.

[20] E. N. Preeth, F. J. P. Mulerickal, B. Paul, Y. Sastri, Evaluation of Docker containers based on hardware utilization, 2015. International Conference on Control Communication and Computing India (ICCC), 2015, str. 697–700.

[21] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas., C. Hauser, J. Domaschka, Cloud Orchestration Features: Are Tools Fit for Purpose ?, IEEE/ACM 8th International Conference on Utility and Cloud Computing, 2015, str. 95-101.

[22] S. Hariri, A. Varma, High-performance distributed computing: Promises and challenges, Concurr. Pract. Exp., Vol. 5, No. 4, 1993, str. 233–238

[23] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, Omega and Kubernetes: Lessons learned from three container-manamagent systems over a decade, Acmequeue, Volume 14, Issue 1, 2016.

[24] A. Tosatto, P. Ruiu, A. Attanasio, Container-based orchestration in cloud: state of the art and chanllenges, 2015 Ninth International Conference on Complex, Intelligent, and Software Intenstive Systems, 2015, str. 70-75.

[25] Solaris Containers. Dostopno na http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html

[26] S. Holla, Orchestrating Docker, Packt Publishing, 2015.

[27] H. Howard, ARC: Analisys of Raft Consensus, University of Cambridge Techincal Report 857 UCAM-CL-TR-857, 2014